# Database State Manipulation in Lazy Functional Programming Languages( Dissertation_全文 )

AUTHOR(S):

Ichikawa, Yoshihiko

# Database State Manipulation in Lazy Functional Programming Languages

Yoshihiko Ichikawa

# Contents

# List of Figures

# List of Tables

# Abstract

A persistent programming language abstracts persistent data access through on-memory data access, making complicated applications easier to build. Therefore, the effectiveness of the language is determined to a certain extent by that of the paradigm and the persistency identification method. This thesis explores a methodology based on the non-strict, statically typed, non-strict functional programming paradigm, which has been proven to be effective at least for formulating queries on complex data. This is the first proposal of a *working* purely functional persistent programming environment using state-transformer monads that incorporates state-based database concepts such as strict update, lazy retrieval, views, integrity constraints, and active rules without compromising purity and non-strictness of the paradigm.

The primary target is Haskell, which is the standard of such programming languages and has been known by its use of a state-transformer monad to handle input/output operations and the type class mechanism to incorporate *ad hoc* polymorphism. The thesis addresses and proposes solutions to the key issues toward making it a *persistent* language.

The proposed methodology is based on the monad of state-transformers. It can encode diverse effects on the state, and amongst others, destructive operations can be coded naturally without disrupting purity of the paradigm. While it also supports the database operations, it complicates programming tasks because of the *single-threadedness*. To lessen this inherent burden of the programming tasks, the proposed method makes use of explicit versioning of the database state, which can be retrieved lazily, even though pri-

mary database state is updated destructively. This capability to handle multiple versions simultaneously naturally extends to support of views, exception handling and "what-if" semantics of execution.

Moreover, the approach identifies persistent roots by their types instead of their string-or or variable-names, which are usually used to identify them. This allows every expression to be typed statically, even when they are constructed using operations manipulating persistent roots. The persistent programming environment also provides programmers with "hooks" to customize primitive operations. Those hooks are suitable for specifying integrity constraints and supporting the concept of active databases.

The prototype has been built by porting a Haskell interpreter, Hugs, to the environment of Texas Persistent Store that support C++-based persistent heap mechanism. Although the current prototype is still at the preliminary level, this dissertation addresses the several related issues by explicitly mentioning the implementation methods.

# Chapter 1

# Introduction

New database applications that require more flexible data structures and more computational power have recently emerged (Jacobs et al. 1995; Flickner et al. 1995; Christophides et al. 1994). The persistent programming system is one of the prominent tools to support the construction of such applications (Atkinson et al. 1983; Atkinson and Morrison 1995; Paton et al. 1996). In contrast to the ordinary database programming style, a persistent programming language extends a volatile programming language so that database access is abstracted by on-memory data access (Fig. 1.1). Therefore, effectiveness of a persistent programming language is determined to a certain extent by that of the underlying programming paradigm, and by that of the adopted method to identify persistent data. Although "what is the best paradigm" is controversial, the non-strict, statically typed,

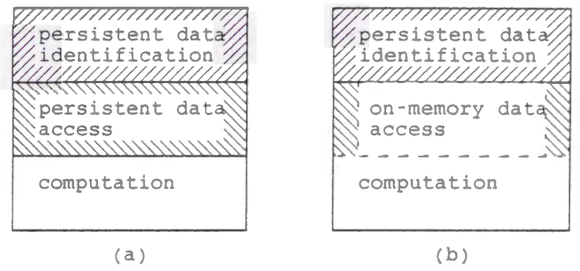



Figure 1.1: Concept of persistent programming; (a) the structure of ordinary database programs; and (b) persistent programming languages abstract persistent data access through on-memory data access.

purely functional programming paradigm naturally exhibits favorable features, summarized as follows:

- Non-strictness (or laziness) [1] allows the users to write mutually-dependent complicated data structures with no additional cost.
- From a user's perspective, static type checking ensures that all the written programs conform to the expected database schema, *viz.*, the collection of types and operators.
- It is easier to make use of mathematical reasoning because of its purity. Since the purity ensures every piece of programs is free from side effects, the potential optimizability based on equational reasoning is higher than that of others even when restricted database queries and general purpose computations are mixed freely (Poulovassilis and Kind 1990; Poulovassilis and Small 1996).

In spite of this potential feasibility of the paradigm, making such a programming language persistent should overcome two key issues with respect to database manipulation:

- Database management tasks include state-based concepts: database update, view maintenance, integrity constraints, and active rules. The paradigm, however, does not have explicit state concepts.
- Prevailing methods to identify persistent roots resort to name- or string-identifiers of them (Dearle et al. 1989; McNally and Davie 1991; McNally 1993; Small and Poulovassilis 1991). Regrettably, name-identifiers would compromise the purity, and string-identifiers would require dynamic type checking.

As for the state-based concepts, the functional programming community has proposed effective techniques, including *continuation*, *linear typing* (Wadler 1990), and *state-transformer monads* (Peyton Jones and Wadler 1993; Gordon 1994; Launchbury and Peyton Jones 1994), to incorporate the state-based programming tasks such as input/output (I/O) operations, and mutable arrays. These techniques are also effective in the context of the persistent functional programming. Indeed, command-continuation, one form of continuation-style, was used by Small (1993) to order database update commands linearly, and linear typing was adopted by Sutton and Small (1995) to ensure the confluence of expressions including side effects. The state-transformer monad was used by Ohori (1990) to formulate the concept of object-identifiers, and its implicit use, i.e., referentially transparent state-based computation, has also been seen in Nikhil (1988) and Nikhil (1990).

[1] Non-strictness does not necessarily imply laziness, nor *vice versa*. However, these terms are often used interchangeably.



This thesis explores the support of state-based concepts needed for database management using the state-transformer monad, without compromising the purity of the functional programming language. The primary target is Haskell, the standard for such programming languages (Hudak et al. 1992; Peterson and Hammonad, eds. 1996; Peterson and Hammonad, eds. 1997), whose notable features include the use of a state-transformer monad to handle I/O operations and the type class mechanism to incorporate *ad hoc* polymorphism (or *function overloading*). A preliminary investigation into the issue is presented in (Ichikawa 1995).

This thesis completes the work by incorporating other important state-based tasks into the methodology. The features of the proposed methodology with respect to the state-based tasks are summarized as follows:

- The database state-transformer monad allows the database state to be updated destructively without compromising the purity of the paradigm.
- Name equivalence on abstract entities can be supported.
- The multiple database versioning reduces the issue of destructive updating and lazy retrieval to a simple concurrency control issue with a single writer and multiple

readers.

- The versioning naturally extends to fully lazy, i.e. just on demand and just once, computation of view values.

The proposed methodology identifies values of persistent roots by their types. The key technique is function overloading. An overloaded function changes its behavior according to its types specified by the context, thus the location of a persistent root is associated with the behavior, having another feature:

- Every expression is typed statically, even if it includes manipulation of persistent root values, without compromising the purity of the paradigm.

For the safety of the location manipulation, the proposed methodology is required to restrict root values to be of ground types, because polymorphic mutable locations cause a typing trouble in Haskell's typing system, as has been seen in the context of ML (Milner et al. 1990; Paulson 1996).

Lastly, it shall be pointed out that the overloaded primitive functions play another role in the proposed methodology. The database access primitives are equipped with "hooks" to customize their behavior. This mechanism with a job queue supports the following features:

- Database designers can customize the behavior of the primitives operators instead of defining dedicated functions to abstract integrity enforcement.
- Transaction-boundary job execution can be incorporated by the ability to access multiple state values simultaneously and the higher-orderness of the paradigm.

This style is flexible enough to support various database manipulation tasks. Among them, this paper exhibits the support for the type extent model of persistency, the management of mutual references, the specification of exception handlers, and the transaction-boundary execution of integrity checkers.



Rest of this chapter explains the emerging advanced database applications, the key concepts of persistent programming, and the features of persistent functional programming.

## 1.1 Advanced database applications

This section reviews the advanced database applications and clarifies their properties. In contrast to traditional database applications such as bank accounting and inventory management, the advanced applications require more complex and flexible data structures, more computational power, and more interaction with external softwares. The first aspect can typically be seen in databases of structured documents (Consens and Milo 1994; Christophides et al. 1994; Christophides et al. 1996). Stored documents are consistently updated by more than one authors, and are compiled into a new document by combining stored components and/or by generating a new component with slight difference from an existing text. Document structures are more flexible than those found in the traditional record-based models of data. Especially, documents should be considered as unstructured, when a query ranges over documents of multiple document types.

The second aspect of the advanced applications, that is, the increased computational power is featured in image database systems, which include non-textual querying facilities. (E.g., Jacobs et al. (1995) and Flickner et al. (1995)). Even the relational data model could manage images as extended primitive values with built-in image processing functions. Such a data manipulation environment, however, might sacrifices the optimizability which is an important property of query languages (Libkin et al. 1996; Seshadri et al. 1997). Integrated computational expressiveness to write almost arbitrary operations as declarative as possible is desirable to write complicated database applications.

The third aspect, the interaction with external softwares, is to some extent one of the current trends of the database programming. Indeed, *SQL Access Group* has specified of

*Call-Level Interface* to for clients accessing SQL-based relational database management systems (Date and Darwen 1997). Although this may be a practical approach to building required applications, the notorious *impedance mismatch problems* (Atkinson and Buneman 1987) still remain between the clients and the servers. Another standardization group, *Object Management Group (OMG)*, has designed a more flexible cooperative concept for integrating object-based information management systems, called *Common Object Request Broker Architecture* (CORBA) (Object Management Group 1997). CORBA "wraps" every object access to encapsulate the behavioral details, the impedance mismatch problems being solved to some extent. However, the every implementation method still have to be written with a certain programming environment, and the effectiveness of the environment affects the overall effectiveness of the system on top of it. General-purpose persistent programming environments that support flexible application construction and naturally embedded database manipulation capability are still in urgent needs.

## 1.2 Persistent programming languages

The above three aspects of the advanced database applications are effectively supported by *persistent programming languages*. The concept of persistent programming languages was proposed more than one decade ago in Atkinson et al. (1983). Persistent programming languages can be considered as computationally complete data definition and/or manipulation languages. The requirements on such languages were summarized by the following principles pointed out in the article:

**persistence independence** : the persistence of a data object is independent of how database applications manipulate the data object, and conversely a fragment of a program is expressed independently of the persistence of the data it manipulates.

**persistence data type orthogonality** : all data objects should be allowed the full range of persistence, that is, any object of any type can be part of a database.



**persistence identification transparency** : how to provide and identify persistence at a language level is independent of the choice of data objects in the language.

Although the above principles clearly characterizes persistent programming environments to develop applications, two more features should be taken into account to apply the environments to developing advanced applications. The first one is the suitability of the languages for prototyping. Prototyping helps specify softwares precisely in an earlier stage of software development, and requires a formal, powerful, and flexible environment. The other feature is the applicability to query processing. Users of a database system include a number of naive users in addition to expert programmers. This means that the system must provide database access at a higher level, where the details of physical database organization are hidden and database expressions may be automatically optimized.

## 1.3 Functional persistent programming languages

To meet the above five criteria, the persistent programming environment ought to be based on a language which has a formal mathematical background, a succinct expression notation, and high optimizability. This dissertation, therefore, selects the functional programming paradigm as the basic background of persistent languages, and discusses the methods to adopt the paradigm for developing advanced database applications. Note that the term, functional programming language, is used to identify diverse kinds of languages based on the concept of function application, and that the languages are classified by typing, purity and strictness.

**Typing** : A language is strongly typed if every expression is type checked and any illegal expression is never evaluated. A language is statically typed if type checking is

performed completely at compile time, and every expression never goes wrong at runtime with no overhead.

**Purity** : A language is purely functional if computation is performed only by applicating functions to values. In other words, there is no side-effect in evaluation.

**Strictness** : A language is strict if arguments of a function is evaluated before the application. Strict evaluation is often called call-by-value, while non-strict one is called call-by-name. Call-by-name evaluation is often called lazy-evaluation, while lazy-evaluation is not the concept regarding strictness, but the strategy to share expressions and their results. The lazy-evaluation does not necessarily mean the call-by-name, but is usually used as the practical method to implement call-by-name evaluation.

In this three-dimensional classification, static typing, purely functional, and non-strict functional paradigm seems most suitable for database application development. Typing is a useful tool for application development and is also important for database applications because types are to programming languages as database schemas to database applications. Static typing ensures that every expression which manipulate databases never goes wrong at run-time. Note also that the arguments in favor of the function paradigm itself have been presented elsewhere. (We refer readers to Hughes (1989) and textbooks such as Bird and Wadler (1988), Holyer (1991) and Davie (1992).)

As described above, database programs must be highly optimizable and must be suitable for query processing. This requires that the programming language itself must be declarative and based upon rigid mathematical theory. Purity ensures that the semantics of expressions are easily captured and that optimization passes are formally defined by equational theory of expressions. Moreover, purely functional languages have a succinct notation, called *comprehension syntax*. The syntax has been proven to be effective in



query construction (Trinder 1991; Buneman et al. 1994). Although the comprehension syntax could be used in impure functional languages, such comprehension expressions should be treated as an abstraction of looping over a collection data, so they are not targets for macro-level optimization because of the potential side effects.

Non-strictness is also desirable for persistent programming. The first point is that non-strictness allows for more flexible recursive data structures. Besides, lazy-evaluation strategy is utilized to implement non-strictness, making it easier to write complicated expressions in a declarative manner.

## 1.4 Outline

Chapter 2 briefly describes the basic features of Haskell, such as expressions, types, I/O model, and class mechanism. Especially the latter two features play the important roles in the proposed method of database manipulation. While the main aim of the chapter is to introduce the language features to unfamiliar readers, it also describes the structure and operations of the *part-supplier* database (Atkinson and Buneman 1987), that is a running example throughout this thesis except Chapter 9.

The remainder of the thesis comprises three parts. The topic of the first part, Chapter 3 through Chapter 5, is the database state manipulation facility of the proposed methods. Chapter 3 explains the database manipulation approach which is based on the state-transformer monad, and clarifies the issues in the straightforward state-based approach. Chapter 4 explains how the persistency of data is specified. Chapter 5 proposes an approach which utilizes multiple versions to relax the imperativeness of the state-transformer approach. This chapter concludes the part with comparison of the proposed method with related work.

Chapters 6 and 7 feature the triggers and active rules in the proposed environment. By making use of the multiple database versions, the class mechanism, and inherent

computational completeness of the language, we have facilitated the environment with "hook" for the primitive database operators. This triggering concept and its application to integrity enforcement, view management and exception handling are discusses in Chapter 6. Chapter 7 generalizes this triggering mechanism so that rule-based computation is allowed in the persistent programming environment. Even though we only take into account the actions associated with a database primitive operations, the multiple database versions make it possible to devise active database features without difficulty and with particular flexibility. The chapter also summarizes the proposed technique and compares it with related work.

The final part of this paper addresses the implementation issues. A prototype environment has been built with a (volatile) Haskell interpreter and a C++-based persistent heap system. The Chapter 8 summarizes the related issues, and explains the current implementation of the prototype. To compare the flavor of the proposed method with other persistent programming languages, Chapter 9 exhibits the train database used as the running example in Paton et al. (1996) to compare several database programming paradigms.

Chapter 10 concludes with summary and future directions.

# Chapter 2

# Background: Notable Haskell Features and the Running Example

The aim of this section is two-fold. The first is to provide an overview of a few notable features of Haskell (Hudak et al. 1992) related to the proposed methodology. Details can be found in tutorials such as Davie (1992) for generic topics, and details on the monadic I/O system can be found in Wadler (1992b), Peyton Jones and Wadler (1993) and Gordon (1994). In this paper, all the lines in program fragments are preceded by > signs for clarity[1]. Note that the Haskell specification described in Hudak et al. (1992) is Haskell 1.2, while the latest specification is Haskell 1.4, on which this paper's work is based.

The other aim is to explain the structure and example tasks on the part-supplier database used by Atkinson and Buneman (1987) to compare miscellaneous languages in the context of database management. This example was also used by Gamerman et al. (1992) to compare the object-oriented approach with other existing ones to programming database applications. This section introduces the Haskell features using the volatile version of this example, and the following sections will give the corresponding persistent version.

[1]Punctuations are also excluded from the program fragments to avoid confusion. "..." is sometimes used to indicate a part which is eliminated from the code to hide cumbersome or implementation-dependent details.

## 2.1 Data types, expressions, and bindings

A type is either an algebraic one or a type synonym. Consider as an example data types for the part-supplier database comprising *Part* and *Supplier*:

- A part is either basic or composite. A basic part has *name*, *cost*, *mass*, *used-by*, and *supplied-by* attributes, and a composite part has *name*, *assembly-cost*, *mass-increment*, *used-by*, and *composed-of* attributes; and
- A supplier has *name*, *address*, and *supplies* attributes.

These objects can be represented by algebraic data types, declared as follows:

```
> data Part = Basic     String Int Int [Part] [Supplier]
>           | Composite String Int Int [Part] [(Part, Int)]
> data Supplier = Supplier String String [Part]
```

where `Part` and `Supplier` (left-hand side) are called *type constructors*, and where `Basic`, `Composite`, and `Supplier` (right-hand side) are called *data constructors*. In the above example, two more type constructors are used: one is the tuple type constructor in `(Part,Int)`, and the other is the list type constructor in `[Part]` and the like.

A type synonym is used to name a type. If we want to name "list of parts" type, the following declaration suffices:

```
> type PartList = [Part]
```

In Haskell, a string is a list of characters, thus the type `String` is defined (internally) as follows:

```
> type String = [Char]
```

The data constructor arguments can be labeled to introduce explicit field names. The above algebraic data types, `Part` and `Supplier`, can be alternatively defined with labeled records as follows:



```
> data Part
>   = Basic    { pName::String,   pCost, pMass::Int,
>                pUsedBy::[Part], pSuppliedBy::[Supplier] }
>   | Composite{ pName::String,   pCost, pMass::Int,
>                pUsedBy::[Part], pComposedOf::[(Part, Int)] }
> data Supplier = Supplier{sName, sAddress::String, sSupplies::[Part]}
```

Field names are also used as *selectors* for selecting fields from algebraic data. The name space of field names is the same as that for functions and variables. The above declarations define nine selector functions in total, where `pName`, `pCost`, and `pMass` are shared by `Basic` and `Composite` values.

The pattern matching also handles labeled records. For example, the following function selects names of basic parts from the given list of part values:

```
> basicParts :: [Part] -> [String]
> basicParts parts = [ n | Basic{pName = n} <- parts ]
```

where `::` is read as "has type."

In connection with this example, we note a few aspects of the language. The Haskell type system permits parametric polymorphism (using a traditional Hindley-Milner type structure like ML), extended with *ad hoc* polymorphism, or *overloading* (using *type classes* described later). Thus the principal type or most generic type of an arbitrary expression can be inferred from the type system, and the first line of the above example that declares the type of the function is optional[2]. Nevertheless, we will often include the type annotations of functions in the remainder of this paper, even when the function bodies are also included. This conventional style is followed mainly for readability.

The above definition uses the *list comprehension syntax* (Wadler 1987) in its right-hand side of the definition. A list comprehension abstracts iteration over a list. In general, a list comprehension takes the following form:

[2] *Ad hoc* polymorphism may result in ambiguous typing. We do not enter into this problem for brevity.

$$[\ e \mid q_1,\ q_2,\ \cdots,\ q_n\ ],$$

where $q_i$ $(1 \leq i \leq n)$ is either of a generator, $pattern_i$<-$e_i$, or a guard, $pred_i$. The generator specifies iteration on the list represented by $e_i$ with $pattern_i$ bound for each element of the list. If the element does not match the pattern, the loop "goes" to the next element. $Pred_i$ specifies that a certain loop on a list element is worth continuing. When the predicate evaluates as *false*, then the loop also "goes" to the next element. The following is another example which computes all the pairing from two lists, `xs` and `ys`:

```
[ (x,y) | x <- xs, y <- ys ]
```

The expression `Basic{pName = n}` in the `basicParts` function has two roles: (1) checking if the element of the list `part` is a basic part, and (2) binding the value of the `pName` field to `n`. This pattern matching is often used to tear apart a labeled record without using selector functions. A field name may be used as the variable name to which the field value is bound. Using this shorthand, the above function could have been simpler like this:

```
> basicParts parts = [ pName | Basic{pName} <- parts ]
```

The extended style of pattern matching is also used to define functions. To illustrate, Fig. 2.1 shows the function that recursively computes the total mass and cost of a part. `fst` and `snd` are library functions that compute the first and second elements of a binary tuple, respectively. `map` and `sum` are also library functions. `sum` computes the sum of a list of numeric values, and `map` applies its first argument to each of the elements of the second argument. The result is the list of the applications.

There is a short-hand notation for field updates for labeled records. For example, a function that updates the address of the given supplier can be coded as follows:

```
> updateSupAddr :: Supplier -> String -> Supplier
> updateSupAddr s newAddr = s{sAddr = newAddr}
```



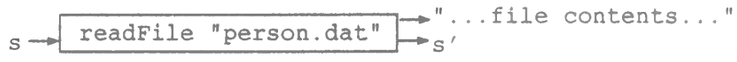
```
> massAndCost :: Part -> (Int, Int)
> massAndCost Basic{pCost, pMass}
>   = (pMass, pCost)
> massAndCost Composite{pCost,pMass,pComposedOf}
>   = (pMass + subm, pCost + subc)
>   where
>     submcs
>       = [ (m * q, c * q) | (p, q) <- pComposedOf,
>                            (m, c) <- [ massAndCost p ] ]
>     subm = sum (map fst submcs)
>     subc = sum (map snd submcs)
```

Figure 2.1: Volatile version of the function to compute the total mass and cost of a part

Note that the "update" is *not* destructive, but constructs a *new* value in which all the field values except `sAddr` are the same as those of `s` and the value of `sAddr` is `newAddr`.

## 2.2 Monadic I/O

An I/O operation using the monadic I/O system is a state transition function. Consider as an example the `readFile` function. This function constructs an I/O operation from a given filename, and the operation is diagrammatically shown like this:




The action takes the I/O state to a pair consisting of the file contents and the new I/O state. From the viewpoint of types, every I/O action returning a value of type `a` is of type `IO a`, where the implementation is hidden from users. For example, the `readFile "person.dat"` is of type `IO String` because the action returns the contents as a string.

There are three combinators associated with the `IO` monad[3]. Fig. 2.2 is a diagrammatical representation of these functions. The simplest function `return` constructs a state-transformer from an arbitrary expression, and the `>>=` composes two state-transformers

[3]The constructor class will be described in Section 2.4.

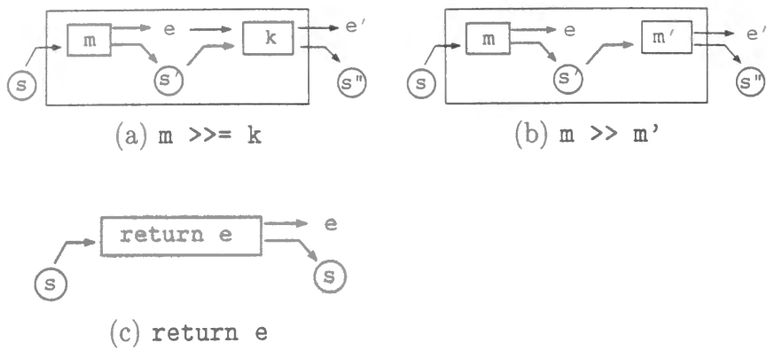



Figure 2.2: Combinators of state transformer monads

while passing the intermediate result from the first to the second. Another one `>>` simply composes I/O actions. The infix operator `>>=` is of type `IO a -> (a -> IO b) -> IO b` which is equivalent to `IO a -> ((a -> IO b) -> IO b)` as the `->` operator is right associative.

## 2.3 Classes and overloaded functions

Haskell uses the *class* mechanism to control operator overloading. For mathematical fundamentals, see Wadler (1989) and Jones (1994b). A class is a family of algebraic types associated with overloaded operators. A class member is called an *instance*, and the operators are called *class operators*. The behavior of an overloaded operator for a specific instance is called a *method*. Consider as an example the definition of the `Eq` class:

```
> class Eq a where
>     (==), (/=) :: a -> a -> Bool
>     x /= y = not (x == y)
```

The second line declares that `==` (equality) and `/=` (inequality) are the class operators. The last line gives the *default method* of `/=`.

The following statements declare two instances of the `Eq` class:



```
> instance Eq Int where
>     x == y = primEqInt x y
> instance Eq Float where
>     x == y = primEqFloat x y
```

Overloaded operators are resolved at run-time (or when possible at compile time). An expression `x == y` is treated as "`primEqInt x y`" ("`PrimEqFloat x y`") if `x` and `y` have type "`Int`" ("`Float`").

Instances can be derived by a Haskell language processor by adding `deriving` clauses in algebraic data type declarations:

```
> data Supplier = Supplier String String [Part]
>       deriving Eq
```

The `deriving` clause specifies that `Supplier` is an instance of the `Eq` class. Note that only instances of predefined classes can be derived and that derived methods are predefined in the language specification. In this case, `Supplier` values are equal when each of their three component values is equal.

## 2.4 Constructor classes and do-notation

The concept of constructor classes was originally proposed by Jones (1994b) to incorporate the monad concept (Moggi 1989; Wadler 1992a) into Gofer, a language akin to Haskell. Instances of a constructor class are general data type constructors instead of (null-ary) data types. The previous section explains `>>=`, `>>`, and `return` as if they were I/O specific operators. Actually, these are the operators of the `Monad` class, declared as follows:

```
> class Monad m where
>     return :: a -> m a
>     (>>=)  :: m a -> (a -> m b) -> m b
>     (>>)   :: m a ->       m b  -> m b
>     m >> m' = m >>= \_ -> m'
```

The list type constructor, `[]`, is also an instance of the `Monad` class, thus the above definition using the list comprehension syntax is equivalent to the following definition in terms of the list type:

```
> basicParts :: Monad m => m a -> m a
> basicParts parts = parts >>= \Basic{pName} ->
>                    return pName
```

The context "`Monad m`" requires that this function should be used with any of the `Monad` instances.

The companion of the `Monad` class is the *do-notation*, which is the generalization of the list comprehension syntax. For example, the above definition is also defined using the notation as follows:

```
> basicParts parts = do Basic{pName} <- parts
>                       return pName
```

The do-notation abstracts computation regarding `Monad`, while the list comprehension syntax abstracts computations of only lists. The following are the simplified translation rules from do-expression to the corresponding one using the monad combinators:

$$\begin{array}{lcl}
\textbf{do}\ \{e\} & = & e \\
\textbf{do}\ \{e;stmts\} & = & e >> \textbf{do}\ \{stmts\} \\
\textbf{do}\ \{p <- e;\ stmts\} & = & e >>= \backslash p -> \textbf{do}\ \{stmts\} \\
\textbf{do}\ \{\textbf{let}\ decllist;\ stmts\} & = & \textbf{let}\ decllist\ \textbf{in do}\ \{stmts\}.
\end{array}$$

To improve the readability, the remainder of this paper will use the do-notation and the monad combinators only for the database and the I/O monads, and will use the list-comprehension syntax for lists.

## 2.5 Expression grammar

Lastly, we include the table of precedences and associativities of the expressions and the part of declarations (Table 2.1).



Table 2.1: Precedence of expressions, patterns, definitions (highest to lowest)

| Item | Associativity |
|---|---|
| simple terms, parenthesized terms | – |
| irrefutable patterns (`~`) | – |
| as-patterns (`@`) | right |
| function application | left |
| `do`, `if`, `let`, lambda(\), `case` (leftwards) | right |
| `case` (rightwards) | right |
| infix operators, prec. 9 | as defined |
| ... | ... |
| infix operators, prec. 0 | as defined |
| function types (`->`) | right |
| contexts (`=>`) | – |
| type constraints (`::`) | – |
| `do`, `if`, `let`, lambda(\) (rightwards) | right |
| sequences (`..`) | – |
| generators (`<-`) | – |
| grouping (`,`) | n-ary |
| guards (`|`) | – |
| case alternatives (`->`) | – |
| definitions (`=`) | – |
| separation (`;`) | n-ary |

# Chapter 3

# Database State Monad

To prevent purity from being disrupted, the proposed methodology utilizes a monad of state transformers to perform referentially transparent update operations. As the I/O mechanism of Haskell is based on the state-transformer monad for the I/O state type, we can introduce another state-transformer monad for the database state. Theoretically, there is no side effect, since the transformers generate a *new* database state value. In practice, on the other hand, state-based operations can be implemented by side effects on the internal state, since the database operations are executed linearly as in an imperative programming language.

We shall mention the non-triviality of this approach here to avoid confusion. An arbitrary state type defines its state-transformer monad. This ensures the generality of the state-transformer approach, but this fact also implies that every property of a monad is determined by the structure of the state type. Therefore, we need to design the state type so that it meets our requirements: destructive updatability, type safety, lazy retrieval, customizable primitives and so on.

The database monad comprises the database state type, primitive operators, state-transformer combinators, and a transaction model. The database state in turn comprises two parts:

- A collection of entities that are represented by introducing surrogates to model their

identities and mutability; and

- A collection of persistent roots that gives the access points of the stored database state.

This section describes only the first part of the state, while the next chapter will describe the second part in conjunction with the safety.

The primitive operators are associated with the `Entity` class, and the entity types are declared by making them the instances of the class. Through this *ad hoc* polymorphism, entity types are discriminated from others. Hence, user-defined polymorphic database operations can be built without difficulty, and the compilation system can prevent programmers from applying database operations to irrelevant types[1]. Although polymorphic *types* may also be instances of the `Entity` class, polymorphic *values* cannot be permanently included in the database state to avoid a problem inherent of mutable locations described in the next chapter. For example, `[a]` may be specified as an instance of the `Entity` class, but only fully instantiated values such as values of type `[Int]` or `[String]` can be stored. Therefore, throughout this paper "entity types" are used to denote such types of storable values instead of "instances of the `Entity` class."

## 3.1 Database state and entities

Let $\Sigma$ be the set of all the entity types in a database, and let $Ref(\sigma)$ and $Val(\sigma)$, respectively, be the sets of all the surrogates and values of type $\sigma$ ($\in \Sigma$). Then the entity-related part of the database state is a collection of $\Sigma$-indexed maps $s_\sigma$. Note that $Val(\sigma)$ is a set of values of Haskell type $\sigma$. A value in $V(\sigma)$ may contain a value of type $Ref(\sigma)$ which refers to other entities directly through pointers and/or indirectly through

[1]Since a database entity is a kind of mutable location, we could use it to implement the *undo-able* mutable variables to reduce the computational order of algorithms. This issue, however, is out of the scope of this paper.



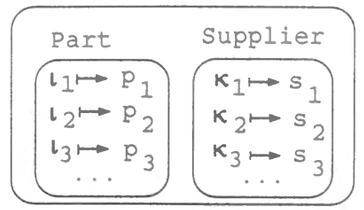



Figure 3.1: Entity part of a database state

entity surrogates. The primitive operations are defined as follows. Let $o \mapsto v$ be a binary association from $o$ to $v$. Then the operational part comprises three operators:

| | |
|---|---|
| $read_\sigma(o)$ | retrieve $o \mapsto v$ from $s_\sigma$; |
| $write_\sigma(o \mapsto v)$ | replace $o \mapsto w$ in $s_\sigma$ with $o \mapsto v$; and |
| $new_\sigma(v)$ | insert $o_\sigma \mapsto v$ into $s_\sigma$ for a new $o_\sigma$. |

Note that due to the restriction applied to the persistent roots types, $\Sigma$ is usually a finite set of ground types.

To illustrate, consider the Part-Supplier database explained in Chapter2. The objects were represented by algebraic data types, or sum-of-product types, declared as follows:

```
> data Part
>    = Basic    { pName::String,   pCost, pMass::Int,
>                 pUsedBy::[Part], pSuppliedBy::[Supplier]}
>    | Composite{ pName::String,   pCost, pMass::Int,
>                 pUsedBy::[Part], pComposedOf::[(Part, Int)]}
> data Supplier
>   = Supplier{sName, sAddress::String, sSupplies::[Part]}
```

The database schema comprises these two types with slight modification. Because stored objects refer to other objects through surrogates instead of direct pointers, the declaration should be modified as follows:

```
> data Part
>   = Basic    { pName::String,          pCost, pMass::Int,
>                pUsedBy::[DBRef Part], pSuppliedBy::[DBRef Supplier]}
>   | Composite{ pName::String,          pCost, pMass::Int,
>                pUsedBy::[DBRef Part], pComposedOf::[(DBRef Part, Int)]}
> instance Entity Part
>
> data Supplier
>   = Supplier{sName, sAddress::String, sSupplies::[Part]}
> instance Entity Supplier
```

Here "`DBRef` $\alpha$" is the Haskell representation of $Ref(\alpha)$, and `Entity` is the class to designate database types. Fig. 3.1 depicts an abstract view of the state of the Part-Supplier database where $\iota_i$ $(i = 1, 2, 3, ...)$ are the surrogates for stored parts and $\kappa_j$ $(j = 1, 2, 3, ...)$ are the surrogates for stored suppliers.

## 3.2 State transformers and basic combinators

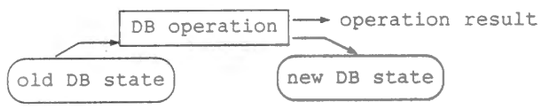



Figure 3.2: Database operation as a state transformer.

Provided that the database state values are of type `DBState`, the type and the monad of database state-transformers may be defined as follows:

```
> data DB a = DB ( DBState -> (a, DBState) )
>
> instance Monad DB where
>    DB m >>= k  = DB (\d -> let (x, d') = m d
>                                DB m'   = k x
>                            in m' d')
>    return x    = DB (\d -> (x, d))
```

where `\x -> e` denotes a lambda abstraction $\lambda x\,.\,e$. These correspond to `IO a`, `>>=`, and



`return` for the I/O state-transformer monad, respectively. Complicated implementation details should be hidden from users in the actual environment, but the above simplified and explicit definitions suffice to show the skeleton of the state-transformer monad. The diagrammatic representation of a database state-transformer is shown in Fig. 3.2. The combinators can be as shown in Fig. 2.2, even though the definitions of the combinators are different from those of the I/O monad.

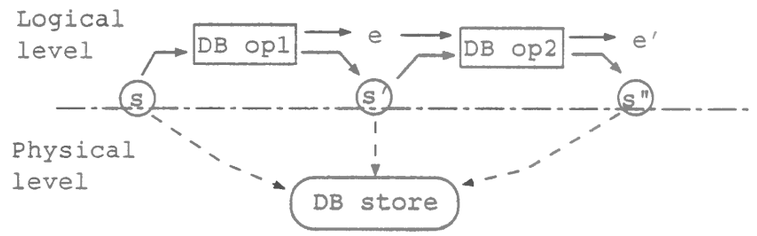



Figure 3.3: Imperative manipulation of database state.

In these figures, database operations are drawn assuming that they construct a new database state value for every step. This is required to ensure the referential transparency. To improve the performance of update operations, however, the database store should be updated destructively. Therefore, a more desirable schematic drawing is shown in Fig. 3.3. Operations are referentially transparent at the logical level, while they may be referentially opaque at the physical level. To make them fully transparent, the state transition should be *hyperstrict*. In terms of Fig. 3.3, this requirement means that when state $s'$ is constructed, the intermediate expression $e$ must have been evaluated so that its subexpressions depending on the the modifiable part of the previous state $s$ are fully evaluated. This restriction is not difficult to enforce, because we only have to ensure it for the built-in primitive operators.

## 3.3 Primitive operations and transactions

The primitive operators are declared in Haskell as follows:

```
>   readDB  :: Entity a => DBRef a       -> DB a
>   writeDB :: Entity a => DBRef a -> a  -> DB ()
>   newDB   :: Entity a => a             -> DB (DBRef a)
```

"`Entity a =>`" specifies the constraint that when these functions are used in a more specific typing context, the variable `a` must be replaced with an instance of the `Entity` class.

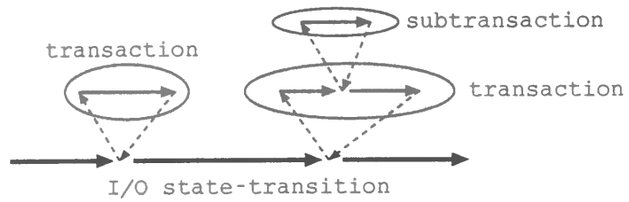



Figure 3.4: Interaction between the I/O world and the database world; the solid arrows at the bottom depict the state-transition sequence in the I/O world, and the other solid ones correspond to database state-transitions.

Every transaction expression is of type `DB` $\tau$, where $\tau$ is the type of the result. This expression is taken to the I/O world by the `transaction` function for execution, or is wrapped up in the `subtransaction` function to be a sub-transaction of another one. These functions are typed as follows:

```
> transaction    :: DB a -> IO a
> subtransaction :: DB a -> DB a
```

The diagrammatical representation of these functions is exhibited in Fig. 3.4. The single-threadedness of I/O operations also ensures that of the database transactions. The modification by a transaction is committed at the end of it by default, whenever all the included database operations are processed successfully.



## 3.4 Examples

To illustrate the usage of the above database primitives, we show a few examples. The first one is a simple query: retrieve basic parts that cost more than $100. Provided that all the surrogates of parts are held in a list bound to `parts`, the query can be coded using the following state transformer:

```
>    mapM readDB parts >>= \partValues ->
>    return [ pName | Basic{pName,pCost} <- partValues, pCost > 100 ]
```

where _ is a wild-card pattern or an anonymous variable. `mapM` is one of the library functions regarding monads. It applies the first argument to the elements of the second argument, and gathers the application results into a list.

As mentioned in the previous subsection, database transactions are "enabled" only after they are taken to the I/O world through the `transaction` function. Hence, the above database query may be executed like this:

```
> selectExpensiveParts parts =
>    putStr "Basic parts that cost more than 100\n" >>
>    transaction (
>      mapM readDB parts >>= \partValues ->
>      return [ pName | Basic{pName,pCost} <- partValues,
>                       pCost > 100 ] ) >>= \names ->
>    putStr (lines names)
```

where `lines` is a predefined function which, given a list of strings, concatenates the strings with a newline character appended to every element of the list, and `putStr` is also a predefined I/O function to construct an I/O action that prints the given string on the terminal screen. For the reader's convenience, we also show the do-notation version of this program:

```
> selectExpensiveParts parts =
>    do putStr "Basic parts that cost more than 100\n"
>       names <- transaction $
>          do partValues <- mapM readDB parts
>             return [ pName | Basic{pName,pCost} <- partValues,
>                              pCost > 100 ]
>       putStr (lines names)
```

The next example updates the addresses of suppliers named "SUP1000" with `newAddr` defined elsewhere, provided that all the surrogates of the stored suppliers can be accessed by the variable `suppliers`:

```
>  transaction (
>    mapM readPairDB suppliers >>= \supPairs ->
>    sequence [ writeDB sid sval{sAddr=newAddr}
>                  | (sid, sval@Supplier{sName}) <- supPairs,
>                    sName == "SUP1000" ]
```

where `readPairDB` is a function defined as

```
> readPairDB s = readDB >>= \v -> return (s, v)
```

Hence "`mapM readPairsDB`" is a function that converts a list of surrogates into a list of surrogate-value pairs. The list comprehension "`[ writeDB .... | ...]`" constructs a list of database actions to perform the required update operations. These actions are combined by the built-in function, `sequence`. The combined action executes the update actions in the order given in the list. Since the update operations are performed destructively, no unnecessary copy is generated.

Consider a slightly more complicated query: retrieve the total mass and cost of a composite part. The volatile version has already been shown in Fig. 2.1. The persistent version can be written as shown in Fig. 3.5, but the query expression is more complicated than the volatile one. Note that the utility function `accumulate` combines the given list of monad actions into a bigger action that returns the list of the results. The imperativeness



```
> massAndCost :: Part -> DB (Int, Int)
> massAndCost Basic{pCost,pMass}
>   = (pMass, pCost)
> massAndCost Composite{pCost,pMass,pComposedOf}
>   = do submcs <- computeSubmcs
>        let subm = sum (map fst submcs)
>            subc = sum (map snd submcs)
>        return (pMass + summ, pCost + sumc)
>   where
>     computeSubmcs
>       = accumulate [ do part   <- readDB p
>                         (m, c) <- massAndCost part
>                         return (m * q, c * q)
>                    | (p, q) <- pComposedOf ]
```

Figure 3.5: "Imperative" query function to compute the total mass and cost of a part

stems from the difference between the structural data access through pattern matching and list comprehension, and the sequential execution of database operators. A more readable definition will be shown later using on-the-fly dereferencing.

# Chapter 4

# Models of Persistence

There are two models of persistence specification: *type extent* and *reachability* models. The proposed approach follows the latter one basically, and supports the former by the help of triggering mechanism described in the next section. Before discussing the details, we briefly overview features of these two models.

In the type extent model of persistence, every database type is associated with a persistent set of entities, or *extent*, of that type. The type extent is maintained automatically by the underlying storage manager: whenever a new entity is created, it is stored in the corresponding type extent. When this model is used to specify persistency, another primitive operation for deletion, say `delDB`, is necessary. Without a certain mechanism to enforce referential integrity, however, a delete operation may result in dangling references that refer to a stale entity.

On the other hand, in the reachability model of persistence, programmers explicitly maintain persistent roots. Values reachable from persistent roots through direct pointers or indirect surrogate references are considered to be persistent. The notable advantage of the reachability model is that it is more flexible and can simulate most of the operations for the type extent model with some additional programming cost. Besides, references dangle less often since entities are deleted only when there is no path from any of the persistent roots. The typical disadvantage of such an approach is its complexity in deleting entities,

that is, all the references to the entity must be modified. If some of the paths should fail to be modified, the database might include an entity that refers to the entity to be deleted. Even though such references never dangle physically, they should be considered to dangle logically since they refer to an obsolete entity. Note that, in the proposed approach, the situation is not so bad, since a database can be updated on a "surrogate-value pair" basis.

At a more abstract level, the persistent roots are just associations from root identifiers to their corresponding root values. The identifiers may be symbols as in Napier88 (Dearle et al. 1989) or strings as in Staple (McNally and Davie 1991). These two approaches, however, do not comply with the purity rule or the static typing rule. Indeed, symbol values cannot be modified without sacrificing purity, and string identifiers require dynamic typing since there is no clue to infer the root types in arbitrary string identifiers. In the approach proposed here, the types of persistent roots are used as root identifiers, and selection of a certain root is implemented naturally through overloaded access functions.

The rest of this section focuses on the support of the reachability model, and also addresses the problem in persistent roots of polymorphic types. Due to this problem, we require every root value to have a ground type. This indirectly requires every permanent values including entities to have a ground type, too. guaranteeing the safety of state capturing and on-the-fly dereference introduced in the next section.

## 4.1 Persistent root declaration

persistent roots are specified using the `PerRoot` class as the `Entity` class is used to define entity types. Provided that two persistent roots for parts and suppliers are maintained for the part-supplier database, the following declarations suffice:



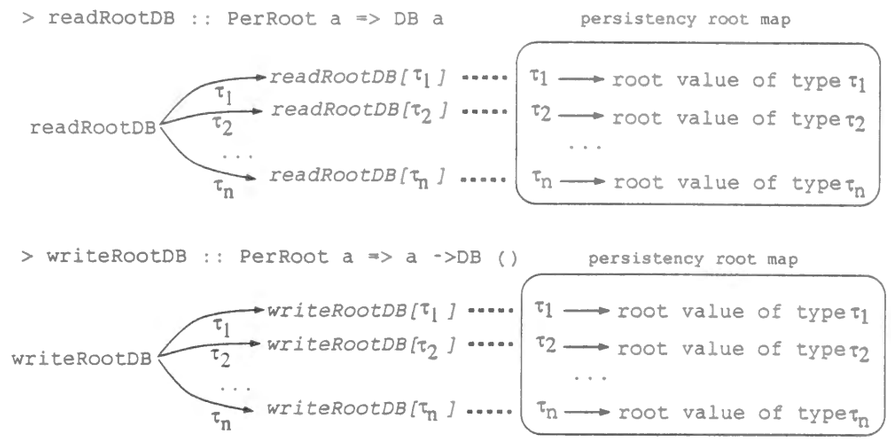



Figure 4.1: Behavior of `readRootDB` and `writeRootDB`

```
> data PartExt = PartExt{parts::[DBRef Part]}
> instance PerRoot PartExt where
>    initValue _ = PartExt{parts=[]}
>
> data SuppExt = SuppExt{suppliers::[DBRef Supplier]}
> instance PerRoot SuppExt where
>    initValue _ = SuppExt{suppliers=[]}
```

The first and the fifth lines declare the root types for parts and suppliers, respectively. The rest of the code fragment declares `PerRoot` instances. The above instance declarations include the specification of initial values in the third and seventh lines. In these cases, they are composed of an empty list. Notice that there is one parameter for the `initValue` method. The parameter is used to define non-materialized views which will be explained in the next section. Throughout this section, these values are always unused.

For each `PerRoot` instance, the underlying storage manager maintains the value of that type. The values of persistency roots are read and written via two overloaded functions:

```
> readRootDB  :: PerRoot a =>      DB a
> writeRootDB :: PerRoot a => a -> DB ()
```

Figure 4.1 diagrammatically represents the behavior of these functions. Since `readRootDB` and `writeRootDB` are overloaded, they have different implementations per instance. In the figure, *readRootDB*$[\tau_i]$ denotes a certain implementation of `readRootDB` for type $\tau_i$, and so does *writeRootDB*$[\tau_i]$. An appropriate implementation is selected automatically through the class mechanism of the Haskell programming language. A context of the root access functions determines the type of the functions, so the type itself determines the corresponding root value location.

## 4.2 Ground type restriction

The Haskell language does not prevent a polymorphic type from being an instance of the `PerRoot` class. For example, `[a]` and `a -> b` may be instances of `PerRoot`. However, supporting a polymorphic root location causes a subtle typing problem that is similar to the one regarding *references* in ML. Connor et al. (1991) also has pointed out that the same trouble arises when subtyping is taken into account. Provided, for example, that `a -> b` be an instance of `PerRoot`, the type be associated with its unique location, and a function `incI` be declared as follows:

```
> incI :: Int -> Int
> incI x = x + 1
```

Then, the following expression would be correctly typed by the type checker:[1]

```
>       writeRootDB incI >>
>       readRootDB       >>= \f ->
>       return (f (2::Float))
```

Note that `>>=` and `>>` are right-associative with the same priority, and that their priority value is lower than that of lambda abstractions denoted by `->` which are also right-associative. Evaluating this expression may lead to a run-time error. In this example,

[1] $e :: \tau$ is an expression that explicitly specifies that $e$ is of type $\tau$.



`writeRootDB` stores `incI` for the root associated with $\forall\alpha.\forall\beta.\alpha \to \beta$, and the stored value is retrieved and used later as a function of type $\forall\alpha.\text{Float} \to \alpha$. The result is not predictable, since the implementation of `Int` is different from that of `Float` in general. The most plausible result is a "segmentation fault" error.

Haskell allows for instance declarations only in the following form:

$$C\ (T\ a_1\ a_1\ \cdots\ a_n),$$

where $C$ is a class name, $T$ is a type constructor name of arity $n$, and $a_i$ $(1 \leq i \leq n)$ are mutually distinct type variables. Programmers may use any specialized forms of the instance type, but the type checker does not discriminate them. As we associate a location with every instance instead of with every specialized form of the instance in the above discussion, the location was shared by all the specialized types. This sharing caused the above problem.

We adopt a simple solution that distinguishes locations of persistent roots at their finest level: every one of them should be associated with a ground specialization of the declared instance type. This section first clarifies the issue in a more formal setting using the technique proposed by Connor et al. (1991), and then mentions the related issues found in ML's references, and in the mutable variable extension of Haskell by Launchbury and Peyton Jones (1994). Then we show how the restriction is enforced by the slightly modified declaration of the `PerRoot` instance.

### More formal view of the problem

The above problem can be clearly seen using the denotational description of storage and the safety condition proposed by Connor et al. (1991). For any location denoted by $i$, written $loc(i)$, three kinds of types are attributed:

- The creation type, written $T_{creation}(loc(i))$;

- The right-hand or stored value type, written $T_{r\text{-}value}(loc(i))$; and
- The view types, $T_{view(j)}(loc(i))$, for every expression, $j$, associated with the location $loc(i)$.

Note that in Connor et al. (1991), $T_{r\text{-}value}(loc(i))$ is called *r-value minimum type*, since record-based subtyping or inclusion polymorphism is treated as the major topic in it. However, we treat parametric typing, and the most general type of a value (inferred from its surrounding context) is assumed unless otherwise explicitly noted. Connor et al. (1991) proposed the following invariant to evaluate the accuracy of static type descriptions:

$$\forall i.\forall j.T_{view(j)}(loc(i)) \leq T_{r\text{-}value}(loc(i)), \tag{4.1}$$

where $\leq$ denotes the parametric subtyping relation; $\tau_1 \leq \tau_2$ holds between the types iff there is a type variable specialization and/or substitution $\theta$ such that $\tau_1 = \theta(\tau_2)$. Intuitively, the stored value must be used as-is or in its more specific form. Note again that, in Connor et al. (1991), $\leq$ denotes record-based subtyping, and that we have exchanged the right- and left-hand sides of the inequality.

Let us view the problem described above using this denotational semantic model of locations. For every `PerRoot` instance $\tau$, the associated storage, say $loc(\tau)$, is of type $\tau$. In the above example, $\tau$ was $\forall\alpha.\forall\beta.\alpha \to \beta$. Because of the specialization rule of typing, `readRootDB` may be of type $\theta(\tau)$, where $\theta$ is a valid specialization of $\tau$. In the above example, this specialized type is $\forall\alpha.\texttt{Float} \to \alpha$. This means that for every view of $loc(\tau)$ the following holds:

$$\forall\tau.\forall\tau'.(\tau' \leq \tau \Longrightarrow (T_{view(readRootDB[\tau'])}(loc(\tau)) \leq T_{creation}(loc(\tau)))), \tag{4.2}$$

where $\Longrightarrow$ represents "implication". Since the root value is read only through this operator, the above inequality implies that

$$\forall\tau.\forall j.T_{view(j)}(loc(\tau)) \leq T_{creation}(loc(\tau)). \tag{4.3}$$



In addition, a similar discussion gives another inequality:

$$\forall\tau.\forall e :: \tau'.(\tau' \leq \tau \Longrightarrow (T_{view(writeRootDB[\tau'] e)}(loc(\tau)) \leq T_{creation}(loc(\tau)))). \tag{4.4}$$

In the above example, $\tau'$ is $\texttt{Int} \to \texttt{Int}$. The value stored in the location is modified only through `writeRootDB`, so this inequality implies

$$\forall\tau.T_{r\text{-}value}(loc(\tau)) \leq T_{creation}(loc(\tau)). \tag{4.5}$$

Finally, the required inequality (4.1) does not necessarily hold. Indeed, this has already been shown by the counter example described above.

We shall mention that the above problem does not arise from the inherent properties of the Haskell type class system. The Haskell language itself does not have the *mutable-location* concept.

**Related issues and techniques**

There are two known methods for avoiding the location problem in statically typed functional programming languages. ML avoids it by using special type variables called *weak type variables*. Type variables appearing in a location type are weak[2]. Unlike usual type variables, weak type variables are not candidates for generalization (or $\forall$ quantification). If this restriction had been applied to our case, the two inequalities (4.2) and (4.4) would have been replaced by the following equalities:

$$\forall\tau'.T_{view(readRootDB[\tau'])}(loc(\tau')) = T_{creation}(loc(\tau')) \tag{4.6}$$

$$\forall\tau'.T_{view(writeRootDB[\tau'] e)}(loc(\tau')) = T_{creation}(loc(\tau')), \tag{4.7}$$

which in combination would have led to the desired equality

$$\forall\tau'.\forall j.T_{view(j)}(loc(\tau')) = T_{r\text{-}value}(loc(\tau')). \tag{4.8}$$

[2]According to the ML terminology, a location is called a reference, and is associated with three operators, `ref`, `!`, and `:=` for creation, reading, and assignment, respectively.

Notice that we use $\tau'$ instead of $\tau$, because the weakness of the type variables requires that every creation type for a location coincide with its usage.

The mutable variable extension of Haskell (Launchbury and Peyton Jones 1994) also adopts the same idea, but implements it by using the monomorphic typing of lambda-bound variables and assigning a special type for the *wrapper*. Although the monomorphic typing is not directly relevant to our current problem, we address it here because it clearly explains why the above problem never occurs in relation to entity locations. Let a polymorphic type $\forall\alpha.\forall\beta.\alpha \to \beta$ be an instance of the `Entity` class:

```
> instance Entity (a -> b)
```

Then the following expression does not incur the problem described above, even though it resembles the above root access manipulation code:

```
> createVar :: DB (DBRef (a -> b))
> createVar = newDB (\x -> bottom)
>
> action = createVar      >>= \v ->
>          writeDB v incI >>
>          readDB  v      >>= \f ->
>          return (f (2::Int))
```

Before discussing further, we shall note the following properties:

1. `createVar` does not create a new location by itself. A new location is created only in a certain sequence of the database state transitions, which in turn is executed only in a certain sequence of the I/O state transitions.

2. Lambda-bound variables are always of monomorphic types according to the Haskell typing rule.

The first property ensures that *a newly created entity location is accessed only through lambda-bound variables*, as shown in the above example, and the second point ensures that *the type of the location is under the monomorphic typing discipline*. These properties



ensure the same typing restriction as was found in ML's weak type variables:

$$\forall\tau'.T_{view(readDB[\tau'])}(loc(\tau')) \;=\; T_{creation}(loc(\tau')) \tag{4.9}$$

$$\forall\tau'.T_{view(writeDB[\tau']\,e)}(loc(\tau')) \;=\; T_{creation}(loc(\tau')). \tag{4.10}$$

Provided that only entity locations are considered, these equalities in combination imply the desired equality as follows:

$$\forall\tau'.\forall j.T_{view(j)}(loc(\tau')) \;=\; T_{r\text{-}value}(loc(\tau')). \tag{4.11}$$

In the above example, the $\tau'$ is `Int` $\to$ `Int` throughout the creation, reading, and writing processes, even though the initial value of the location is `\x -> bottom` whose principal type is $\forall\alpha.\forall\beta.\alpha \to \beta$. In other words, this lambda abstraction is treated as if it were of type `Int` $\to$ `Int` at the creation time.

Now consider another slightly modified example:

```
> action = createVar      >>= \v ->
>          writeDB v incI >>
>          readDB  v      >>= \f ->
>          return (f (2::Float))
```

Using this function incurs a *typing* error instead of a *run-time* error. As noted in the second property above, the location type carried by `v` and `f` must be unifiable without generalization. This requires that `Int` $\to$ `Int` be unified with `Float` $\to \beta$, thus a typing error occurs as expected.

Lastly, we shall point out that there is *no* "wrapper" for the database monad. In general, a state-transformer monad abstracts *computation processes*, while ordinary terms abstracts *values*. The difference also requires that a state-transformer monad must be facilitated with an *evaluator* which executes the abstracted computation process. Launchbury and Peyton Jones (1994) facilitate their lazy state-transformer monad with `runST`, which executes or "wraps" a state-transformer to produce the result value:

$$\texttt{runST}::\forall a.(\forall s.\texttt{ST}\ s\ a) \;\to\; a.$$

We might have facilitated the database monad with an executor like

$$\texttt{runDB}:: \forall a.\texttt{DB}\ a \rightarrow a.$$

However, this gives rise to two problems. The first problem is the one that we have been discussed so far. If we allow a programmer to use the wrapper, (s)he could *name* the entity location with its polymorphic type like this:

```
> createVar' :: DBRef (a -> b)
> createVar' = runDB (newDB (\x -> bottom))
```

Since this function would assign a polymorphic type to the location, we would face the same problem as discussed above. Another problem is loss of linearity of the database state thread. To ensure the *well-definedness* of the database state, we must also ensure the linearity of all the related actions. An example of non-linear expression would be:

```
>     let v = createVar'
>     in (runDB (writeDB v incI), runDB (readDB v))
```

The result depends on the order of evaluation of the tuple elements. Because of the first property of the database monad and the fact that the I/O monad does not have a wrapper either, we can ensure the linearity of the database monad. Note that the technique proposed by Launchbury and Peyton Jones (1994) avoids this illegal "capture" of locations by assigning a special type to `runST`[3]. Briefly speaking, their technique never allows a location to be used outside the local state-transition sequence where the location was created.

## 4.3 Avoiding illegal root manipulation

The second technique is valid only if locations are created and used in a lambda abstraction. On the other hand, the locations of persistent roots are implicitly generated *in*

[3]The type is not a Hindley-Milner type, because the quantifiers are not all at the top level.



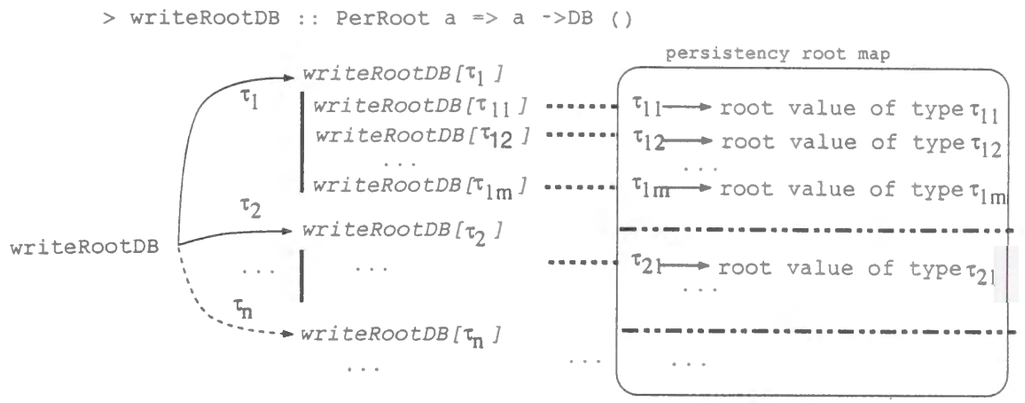



Figure 4.2: Behavior of `writeRootDB`

*advance*. Our proposed solution thus imposes a restriction that stored root values should be of ground type. Note that this does not necessarily prevent database designers from using polymorphic root instances. Instead, even though a polymorphic type is declared as an instance of `PerRoot`, only its ground specializations are treated as the types of the stored roots locations.

Following this setting, every branch shown in Fig. 4.1 has one or more of sub-branches according to the ground specialization of an instance type (Fig. 4.2). *WriteRootDB*$[\tau_i]$ denotes the branch of `writeRootDB` for $\tau_i$ that is an instance of the `PerRoot` class. *WriteRootDB*$[\tau_{ij}]$ is a one-step-further refinement of *writeRootDB*$[\tau_i]$ where $\tau_{ij} = \theta(\tau_i)$ for some ground specialization, $\theta$. The appropriate implementation method is automatically selected by the class mechanism of Haskell.

This restriction requires that the `PerRoot` class is declared in a slightly different way:

```
> class Ground a => PerRoot a where ...
```

where "`Ground` $a$" gives the context of this class declaration, and requires that only if $a$ is an instance of `Ground`, can it be an instance of `PerRoot`. This implies that `readRootDB` and `writeRootDB` implicitly require their related types be instances of `Ground`. Therefore,

an appropriate sub-branch in Fig. 4.2 is selected automatically.

It is trivial to see that this restriction is sufficient to ensure inequality (4.1). Indeed, since all locations are of ground types, inequalities (4.2) and (4.4) are reduced to simple equalities respectively as follows:

$$\forall \tau'.T_{view(readRootDB[\tau'])}(loc(\tau')) \;=\; T_{creation}(loc(\tau')) \tag{4.12}$$

$$\forall \tau'.\forall e :: \tau'.T_{view(writeRootDB[\tau']\, e)}(loc(\tau')) \;=\; T_{creation}(loc(\tau'))), \tag{4.13}$$

which in combination imply the desired equality

$$\forall \tau'.T_{view(\tau')}(loc(\tau')) \;=\; T_{\tau\text{-}value}(loc(\tau')). \tag{4.14}$$

We also use $\tau'$ instead of $\tau$ to indicate that the type is restricted to being a ground specialization of a certain root type $\tau$.

Although how to enforce this restriction is an implementation issue, we describe the basic strategy here to make the above intuitive explanation more concrete. Instances of `Ground` are categorized into two groups. The first group simply includes ground types like `Bool`, `Char`, and `Int`:

```
> instance Ground Bool
> instance Ground Char
> instance Ground Int
>   ...
```

The other group comprises (algebraic) data type constructors with arity of more than one. Remember that the Haskell class mechanism allows an instance declaration to have its specific context. For instance, the `Ground-[a]` instantiation is declared like

```
> instance Ground a => Ground [a]
```

where "`Ground` $a$" is the context of this instance declaration, which is to say that iff a type $a$ is an instance of `Ground`, then so is the type `[`$a$`]`.



The first group gives the base cases of the structural induction, and the second one gives the induction step. Henceforth, it suffices to ensure that all the predefined and user-defined algebraic data types are correctly made instances of the `Ground` class. Note also that it is impossible for users to give their own instance declaration for the class: Haskell prohibits overwrapped instance declarations[4], and allows only instances to be declared in their most general forms[5]. Any attempt by users to declare their own `Ground` instances invokes an error during compilation.

The above restriction requires that a user be careful in declaring the persistent root types. For example, the user can declare that $\forall\alpha$.`Bag` $\alpha$ is an instance of the `PerRoot` class, but (s)he must now declare it as follows:

```
> data Bag a = Bag [a]
>
> instance Ground a => PerRoot (Bag a)
```

As pointed out by Jones (1994b), this may infer a confusing context. To illustrate, consider a function defined like this:

```
> incBag x = readRootDB >>= \(Bag s) ->
>            writeRootDB (Bag (x:s))
```

This is a database operation that adds an element to a persistent root containing a "bag" value. Because of the restriction described above, the inferred type of this function is

```
> incBag ::  Ground a => a -> DB ()
```

instead of the more intuitive declaration:

```
> incBag ::  PerRoot (Bag a) => a -> DB ()
```

This is an inherent property of the Haskell language, and we cannot avoid it.

[4] Two instance declarations, $C\ \tau_1$ and $C\ \tau_2$, overwrap, if $\tau_1$ and $\tau_2$ are unifiable.

[5] Instance declaration $C\ (T\ a_1\ \cdots\ a_n)$ requires all the type expressions, $a_i$ $(1 \leq i \leq n)$, to be distinct type variables.

Finally, we must mention that the `Ground` instantiation does not require any special technique for its implementation. To support the language specification, every Haskell processor has already been equipped with an automatic instantiation mechanism in a particular compilation phase. Modifying this phase so that the above restriction is automatically generated for all algebraic data types is straightforward[6].

[6]Exceptions are constructor variables.

# Chapter 5

# Lazy Retrieval and Imperative Update

As pointed out in Section 3.4, even complicated recursive functions must resort to step-by-step, or imperative, execution in the core of the monadic database manipulation. The source of this imperativeness is the lack of one-the-fly dereferencing from a surrogate. This complication was necessary for the referential transparency not to be compromised, but it sacrificed the declarativeness and terseness of Haskell programs.

This issue is closely related to the parallel nature of Haskell. Because of its declarativeness, the language naturally exhibits parallelism in execution. For instance, the two subexpressions in $e_1$ + $e_2$ may be evaluated in three different orders: $e_1$ then $e_2$, $e_2$ then $e_1$, or $e_1$ and $e_2$ in parallel. The third, parallel execution, does not affect the result of execution because of the referential transparency. Moreover, in non-strict programming languages, expressions are evaluated on demand. Evaluating an expression constructs the datum that represents the computation process instead of the one that represents the result of computation. This computation datum is often called a *closure*. Thus, expressions of non-strict functional languages are evaluated by two interleaving steps: (1) constructing a closure, and (2) performing (or reducing) a closure. Even though the evaluation is performed in a single-CPU system, the underlying evaluator must include the scheduler and the evaluator of closures.

Since Haskell is parallel in nature as explained just above, we must control the execution of closures at least so that the on-the-fly dereference through surrogates does not break the referential transparency of the programming system. Although simply implying a certain order of evaluation might solve the issue, it would depend on the particular system implementation and hence reduce the portability of the system. In this section, therefore, we pursue a method that is easy to port and does not affect the language specifications of Haskell.

The basic idea is the same as the "notorious" *state capture* which may disrupt the linearity of a state-transformer monad by introducing a function like this[1]:

```
getState = DB (\s -> (s, s)).
```

To ensure the linearity, we have two choices:

1. To invalidate the state capture operation, or

2. To duplicate the state value.

Thus this issue is reduced to a transaction control issue among one *writer* and multiple *readers*. The first choice implies that the writer gets an exclusive lock on the state to ensure linearity. The other choice leads to the mechanism called *multiple versions concurrency control* (Bernstein et al. 1987). In this mechanism, a transaction manager keeps track of modification histories of data items. When a data item is modified, the history is updated so that the old value may be retrieved later. Even when a transaction issues a read request for an older value, the request is successfully processed if an appropriate old value is found in the modification history. The technique proposed in the following sections uses an idea similar to this transaction mechanism, but uses explicit version-controlling primitives to make it easier for programmers to handle multiple versions simultaneously.

[1]Since the database state-transformer monad is defined by an abstract data type, users are prohibited to define this functions.



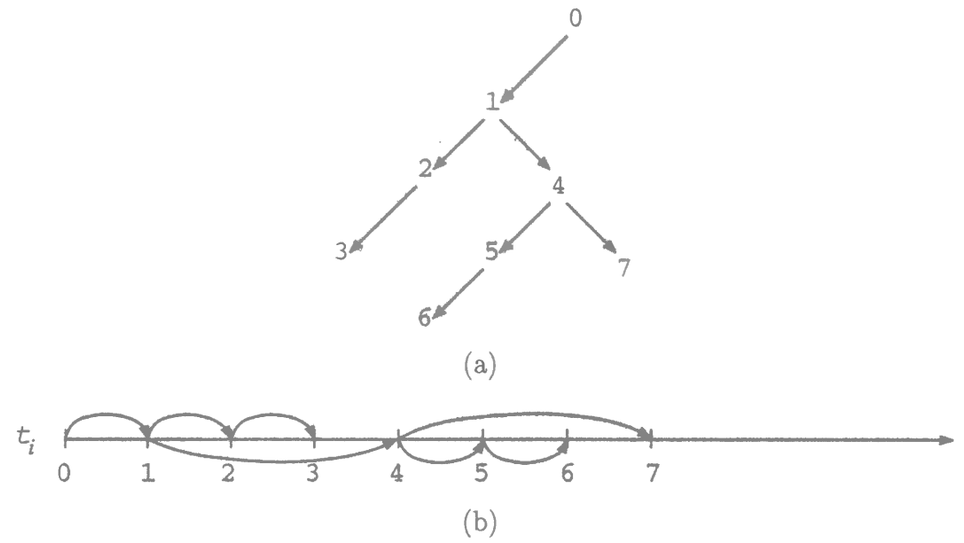



Figure 5.1: Version generation relationships: (a) shows the version generation relationships in a tree form, and (b) shows an alternative view of the history plotted in the time domain.

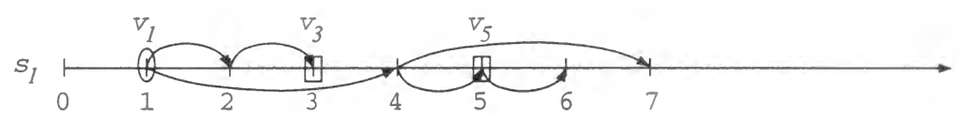



Figure 5.2: History of an object: The circle designates the birth of the object, and two squares are put on the points where the object is modified. The values at the non-marked places after the birth are inherited according to the version history.

Every transaction is a *writer* that may modify the database state, and expressions that retrieve data from the captured state values are *readers*. All the *writers* are linearly ordered, but the *readers* can access captured state values on-the-fly.

## 5.1 State capture

Fig. 5.1 shows a conceptual view of of a database state history. In Fig. 5.1 (a), the number $i$ represents a version generated at time $t_i$. Fig. 5.1 (b) shows the same version history plotted in the time domain. Similar plotting can be used to exhibit an entity

history as shown in Fig. 5.2. The entity is created at $t_1$. The entity is modified at $t_3$ and $t_5$, respectively with $v_3$ and $v_5$. The values at other points are inherited according to the version generation relationships. For instance, the entity value is $v_5$ at $t_6$, and $v_1$ at $t_7$. Dereferencing the entity value for $t_0$ incurs an error. Remember that while objects are "born" explicitly, they "die" implicitly when they become garbage.

At any point of a database state-transformer execution, the current state can be captured by the action `getDB`. Provided that `Database` is the abstract type that represents the captured database state, the operation is typed as follows:

```
> getDB :: DB Database
```

Since `getDB` implies the existence of readers, it gets a read lock on the current database state. The details of the captured value depends on the implementation technique, but it is argued conceptually that the value can be considered to represent the entire database state. Thus, we can restore the previously captured state by the following function:

```
> restoreDB :: Database -> DB ()
```

Every writer's operation should see if it conflicts with a read lock on the expected version. Whenever the read lock is obtained by a reader expression, the writer does not modify the state itself, but generates a new state value. This conservatism is required to ensure that all the readers and writers are failure-free. While a lock on a version is explicitly got by `getDB`, the lock should be released by a garbage collector. When an expression that holds a lock on a certain version becomes garbage, the garbage collector can release the lock.

In addition to these operators, the original state of a transaction is useful for functions that manipulate more than one database versions. For now, we only show the primitive operator to access the original state:



```
> getOrigDB :: DB Database
```

This operator does not increase the cost of database state manipulation. Since the original database state is always kept (at least virtually) intact during a database transaction to support the rollback operation, even when this operator is executed in a transaction, the number of database state values required in the transaction does not increase.

## 5.2 On-the-fly location access

Before showing on-the-fly access operators to the captured state, we shall mention again that the entity locations do not incur any typing trouble. An entity location is accessed only through traversal from a persistent root value or is newly created by `newDB`. The first case is not dangerous because of the ground type restriction noted in the previous chapter, and the newly created entity is manipulated consistently because of the monomorphism restriction put on the lambda-bound variables. Moreover, the single-threadedness is always satisfied, since there is no *wrapper* that runs a state-transformer on-the-fly.

There are two primitive operators for retrieving data from the captured database state lazily:

```
> readRef  :: Entity a  => Database -> DBRef a -> a
> readRoot :: PerRoot a => Database -> a
```

The former dereferences a surrogate in the given version, and the latter retrieves a persistent root from it.

Now that on-the-fly dereference is permitted, we consider the query at the end of the last section again: retrieve the total mass and total cost of a composite part. This query can be written as follows:

```
> massAndCost :: Database -> Part -> (Int, Int)
> massAndCost db Basic{pCost,pMass}
>   = (pMass, pCost)
> massAndCost db Composite{pCost,pMass,pComposedOf}
>   = (pMass + subm, pCost + subc)
>   where
>     submcs
>       = [ (c*q, m*q) | (p, q) <- pComposedOf,
>                        (m, c) <- [ massAndCost(db) (readRef(db) p) ]
>     subm = sum (map fst submcs)
>     subc = sum (map snd submcs)
```

The differences from the function shown in Fig. 2.1 are that `readRef` is used and that `db` is passed to `massAndCost`.

## 5.3 The "what-if" semantics

Multiple versions allow us to define an action that is equivalent to `return` except that it forces the surrounding (sub-)transaction to restore the original state:

```
> markAbortDB :: a -> DB a
```

Note that this function affects only the *current* state value left at the end of the transaction. Even when the original state is restored, the computation process performed in the transaction is still valid. This property and the explicit versioning support the "what-if" scenario effectively. For example, suppose that the total mass and cost of a particular composite part have to be evaluated under a hypothesis that a certain database update is performed. Section 5.2 has already defined `massAndCost`, which computes the cost and mass of a particular part, thus the following simple function suffices:



```
> whatIfCandM :: DB () -> DBRef Part -> IO (Int, Int)
> whatIfCandM updateParts pid
>    = transaction (
>         do updateParts
>            db <- getDB
>            markAbortDB (massAndCost(db) (readRef(db) pid)) )
```

This function can be generalized so that the hypothetical update is performed on a particular database state. The key technique is to use `restoreDB`, which makes the given database state current. By temporarily making the given state current and then discarding the modification applied to the state, the "what if" scenario is written as

```
> hWhatIfCandM :: Database -> DB () -> DBRef Part -> IO (Int, Int)
> hWhatIfCandM db updateParts pid
>    = transaction (
>         do restoreDB db
>            updateParts
>            db <- getDB
>            markAbortDB (massAndCost(db) (readRef(db) pid)) )
```

## 5.4 Non-materialized views

One of the important features of database management systems is the support of views, or derived values. Views can be categorized into two types: materialized views, and non-materialized views, where view materialization is the computation of view values. The materialized views are associated with their pre-computed values, so the maintenance of them reduces to an integrity enforcement issue, which will be described in the next chapter. On the other hand, reading a non-materialized view invokes the computation routine associated with the view. The former requires more storage, while the latter incurs a computation overhead. Which to use should be decided on a case-by-case basis. In the proposed Haskell-based database management environment, both materialized and non-materialized views are supported. The remainder of this section describes the support of

```
> data BParts = BParts {basicParts::[DBRef Part]}
> instance PerRoot BParts
>   initValue db
>     = let PartExt{ parts } = getRoot(db)
>       in BParts{ basicParts =
>            [ pid | pid     <- parts,
>                    Basic{} <- readRef(db) pid ] }
>   isView _ = True
>
> data CParts = CParts {compositeParts = [DBRef Part]}
> instance PerRoot CParts
>   initValue db
>     = let PartExt{ parts } = getRoot(db)
>       in CParts{ basicParts =
>            [ pid | pid         <- parts,
>                    Composite{} <- readRef(db) pid ] }
>    isView _ = True
```

Figure 5.3: Declarations of non-materialized view roots.

non-materialized views only. The materialized views will be discussed later in Section 6.4.

To illustrate, we define two views of the stored parts: one for basic parts and the other for composite parts. The idea of implementing non-materialized views is straightforward. Remember that the initial values of persistent roots are specified by the `initValue` method. This means that when there are no explicitly stored root values, this method is automatically invoked by the underlying system. Hence, if we define the view definition in the right-hand side of the method, it becomes the default value of the root. Thus, providing a *switch* to prohibit the modification of the root values is enough to define non-materialized views. Following this idea, the non-materialized views for the basic and composite parts can be specified as shown in Fig. 5.3.

The argument supplied to `initValue` is the current database state. The right-hand side of the method computes the initial value using the current database value. The `isView` method works as the switch to control the updatability of the root value. Only if



it evaluates to `False`, the root update action `writeRootDB` is allowed to be executed on this root. Otherwise, the action invokes a run-time error. `isView` is typed thus

```
>     isView :: [a] -> Bool
```

where `a` is assumed to be the type of the persistent root. Because of the restriction imposed by the class mechanism of Haskell, the type of the persistent root must appear in the type signature of the class method. In the underlying implementation, the actually supplied value is `[]`, and the type consistency is manipulated by an explicitly specified type signature.

In contrast to the materialized views, non-materialized views are dynamically computed on demand. The notable advantage over the materialized views is the minimal cost of view maintenance for programmers and the systems, but the view values may be repeatedly computed. There is a simple solution to this problem. As the type signature of the `initValue` implies, the view values are computed based on the database state supplied as the first argument. If the computed value is cached before it is returned to the user code requiring the view value, later access to the same view can be performed simply by fetching the value in the system cache associated with the current database state. Thus, the internal view access code may look like this:

```
do b <- ''check if the current state caches the required value''
   if b && ''cached value is that for the current state''
    then return ''the cached value''
    else do db <- getDB
            let val = initValue db
            ''store val in the cache''
            return val.
```

To keep the cache contents consistent, the view values are recomputed after the database state has been updated. The above pseudo-code uses `getDB`. This locks the current version of the database state, thus the lazy computation of view values is performed safely.

## 5.5 Formalization and relationship to array freezing

This subsection clarifies the difference between the database versioning and array freezing (Launchbury and Peyton Jones 1994)[2] by showing the formalization of the database versioning using the monad of "lockable array"-transformers. The monad is a variation of the monad of array transformers (Wadler 1992c). Here we consider array subscripts as surrogates and associated array values as associated entity values. A database state is represented by a pair consisting of an array and an identifier generator of type $(Arr, Integer)$, where $Arr$ is the type of arrays with indices of type $Integer$ and values of type $Val$. For brevity, types and root management are not treated here.

First consider the database monad as an array-transformer monad. The monad can be defined as follows:

$$\begin{array}{lcl}
\textbf{type}\ DB\ a & = & State \to (a, State) \\
\textbf{type}\ State & = & (Arr, Integer) \\
\texttt{return} & :: & a \to DB\ a \\
\texttt{return}\ a & = & \lambda x.(a, x) \\
(\texttt{>>=}) & :: & DB\ a \to (a \to DB\ b) \to DB\ b \\
m \texttt{>>=} k & = & \lambda x.\textbf{let}\ (a, y) = m\, x\ \textbf{in} \\
 & & \quad \textbf{let}\ (b, z) = k\, a\, y\ \textbf{in} \\
 & & \quad (b, z).
\end{array}$$

If *index* and *update* are the read and destructive write operations for $Arr$, the basic operations are defined like this:

$$\begin{array}{lcl}
\texttt{newDB} & :: & Val \to DB\ Integer \\
\texttt{newDB}\ v & = & \lambda(x, g).(g, (update\ g\ v\ x, g + 1)) \\
\texttt{readDB} & :: & Integer \to DB\ Val \\
\texttt{readDB}\ i & = & \lambda(x, g).(index\ i\ x, (x, g)) \\
\texttt{writeDB} & :: & Integer \to Val \to DB\ () \\
\texttt{writeDB}\ i\ v & = & \lambda(x, g).((), (update\ i\ v\ x, g)).
\end{array}$$

For this monad to be single-threaded, the `readDB` operation must be executed *hyperstrictly*: before returning the operation result, it must compute $index\ i\ x$. Thus the

[2] Array freezing generates a standard Haskell array from a mutable one that is directly updatable in a state-transition sequence.



`readDB` operator should be a built-in one that has this property. If Wadler's **let!** notation (Wadler 1990) were used here, the definition would be like this:

$$\begin{array}{lcl}
\texttt{readDB}\ i & = & \lambda(x, g).\,\textbf{let!}\ (x)\ v\ =\ index\ i\ x \\
 & & \textbf{in}\ (v, (x, g)),
\end{array}$$

where **let!** implies that $v$ is computed hyperstrictly with respect to $x$; that is, every component of $v$ depending on $x$ is evaluated before commencing the evaluation of $(v, (x, g))$.

Let us define the `getDB` and `readRef` operations following the array-freezing approach. Suppose that *dup* be an operation that duplicates an array, then the state-capturing and on-the-fly dereferencing operations would be defined as follows:

$$\begin{array}{lcl}
\texttt{getDB} & :: & DB\ State \\
\texttt{getDB} & = & \lambda(x, g).(dup\ x, (x, g)) \\
\texttt{readRef} & :: & State \to Integer \to DB\ Val \\
\texttt{readRef}\ x\ i & = & index\ i\ x.
\end{array}$$

Like `readDB`, `getDB`must be hyperstrict: before returning the operation result, it must compute $dup\ x$. Again with the **let!** notation, the defintion would be thus

$$\begin{array}{lcl}
\texttt{getDB} & = & \lambda(x, g).\,\textbf{let!}\ (x)\ x'\ =\ dup\ x \\
 & & \textbf{in}\ (x', (x, g)).
\end{array}$$

This is the basic idea of array freezing[3]. In this scheme, even if the state array is not going to be modified in future, it is duplicated. In addition, *dup* may generate multiple copies of a single array, even when a single copy suffices.

On the other hand, the database versioning simply locks the state and duplication is performed only when necessary. Let us suppose that locking, unlocking, and lock test functions on $Arr$ are available:

$$\begin{array}{lcl}
lock, unlock, clear & :: & Arr \to Arr \\
locked & :: & Arr \to Bool,
\end{array}$$

where *lock* (*unlock*) increments (decrements) the lock counter of the given array and *clear* resets the lock counter. Then the database versioning operations can be defined as

[3] The type of *mutable* arrays is different from that of *immutable* ones in Haskell. The idea presented here can be applied to the actual environment with slight modification, though.

follows:

$$
\begin{array}{lcl}
dup' & :: & Arr \rightarrow Arr \\
dup'\ x & = & \textbf{if}\ locked\ x\ \textbf{then}\ clear\ (dup\ x)\ \textbf{else}\ x \\
\texttt{newDB}\ v & = & \lambda(x,\ g).(g,\ (update\ g\ v\ (dup'\ x),\ g+1)) \\
\texttt{readDB}\ i & = & \lambda(x,\ g).(index\ i\ x,\ (x, g)) \\
\texttt{writeDB}\ i\ v & = & \lambda(x,\ g).((), (update\ i\ v\ (dup'\ x),\ g)) \\
\texttt{getDB} & = & \lambda(x,\ g).\textbf{let}\ x' = lock\ x\ \textbf{in}\ (x',\ (x',\ g)) \\
\texttt{readRef}\ x\ i & = & index\ i\ x,
\end{array}
$$

where *dup'* tests the given array and duplicates the array only if it is already locked. The implementation has an overhead for lock testing, but unnecessary duplication never occurs in database state-transition sequences. Indeed, hyperstrictness is required in evaluating `readDB` as described above. The `getDB` operator, however, does not require duplication, and postpones the duplication until the `newDB` or `writeDB` operators actually require the copy of the state value.

The above formalization does not capture the reachability model of persistence. This aspect is not easy, even if it is possible, to incorporate in the above monadic formalization, since the attempt requires a formal model of "heap" systems. However, for clarifying the difference between array-freezing and versioning, the above formalization suffices.

## 5.6 Related work on database state updating

### 5.6.1 Lazy state-transformers

The present approach is based on imperative state transformers, but there is another approach based on the *lazy state-transformers* or *shadow paging* technique (Argo et al. 1990; Nikhil 1990; Nikhil 1988; Maier and Stein 1987). In that approach, the database state is immutable, and it is updated by tearing apart the old value and constructing a new one. To reduce the cost of state construction, unchanged parts of the new and old values are shared by backward pointers.

Fig. 5.4 shows the database state structured in a tree form. The left tree is the old state with *t1* as its root. The right tree exhibits the new state after the node labeled $n$



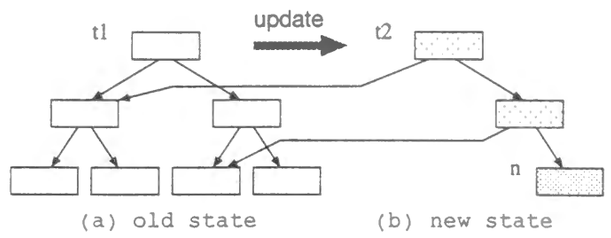



Figure 5.4: Lazy state transformers with database state in a tree form: (a) *t1* is the root of the old state, and (b) *t2* is that of the new one. $n$ is the newly created node by the update operation. The path from the root to the updated node $n$ is also duplicated so that the old state is intact.

has been updated. Notice that the old state is completely intact, and the newly created state include three new nodes: one for the updated node, and two for the path from the root to the updated node.

The cost of duplicating the path to the updated node could be reduced by modifying it directly instead of creating a new node if possible. This condition can be easily checked if a reference count garbage collector is utilized to implement the persistent pages.

The shadow paging chooses duplication by default, and only when applicable, is an in-place update performed. On the other hand, the methodology proposed in this paper modifies the current version destructively by default, and only when it must be kept for later use, a new version is generated.

The difference in performance obtained with these approaches is not clear and must be investigated further. At least for successive modifications of the database state, however, the imperative update mechanism can be executed more efficiently. Moreover, even if locking and updating are performed alternately, the common part of versions may be shared as in the lazy state-transformer approach.

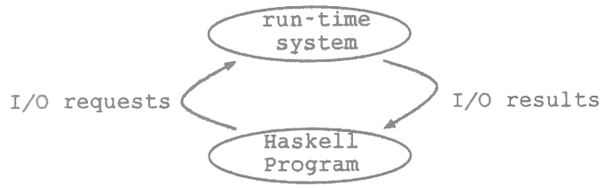



Figure 5.5: Stream-based communication for a program and an external run-time system

### 5.6.2 Persistent streams

Another model of the mutable state uses persistent streams (McNally and Davie 1991; Hammond et al. 1993). While Haskell 1.3 and 1.4 specify monadic I/O operations, the older language definition, Haskell 1.2 (Hudak et al. 1992), adopted dialogue-based I/O operations. In this model, a Haskell process is connected with an external run-time process through two command channels (Fig. 5.5). The left channel is used to send operation requests from the program to the run-time system, and the other is used by the program to receive the results of the processed requests.

Staple (McNally and Davie 1991) extends this I/O system to support persistent streams. A persistent stream is similar to a file except for the contents. While a file contains a list of characters, a persistent steam stores a value of type `any`, which has its type annotation in its representation. Programmers can convert a normal value into the corresponding value of type `any` through `mkany`. The reverse conversion is performed through `coerce`. In this conversion process, the dynamic type checking is performed at run-time to see if the expected type is equal to the type associated with the `any` value. The requests in a program are organized in a lazily generated list, and the run-time process works as a sequential transaction manager that processes the incoming requests in the order in the list. Hence, the side effects by the run-time process does not ruin the referential transparency of the language.

In spite of the difference between streams and monads, the persistent stream can support the approach described here. Indeed, Peyton Jones and Wadler (1993) show the equivalence between the stream-based I/O system and monad-based one. The difficulty arises, however, in the implementation of the database state versioning. Since the external run-time system manipulates all the input operations including file and terminal I/O operations, creating a new version leads to duplication of the complete system environment. To support the versioning, the persistent stream model must be extended so that the run-time process is composed of multiple request handlers to which requests are appropriately dispatched.



Another difference between Staple and the proposed approach exists in the typing principle. As explained above, Staple is a strongly typed language, while our proposal adheres to static typing, which is the basic typing principle of Haskell. Although strong typing is more flexible, it lessens the reliability of database programs. A persistent stream is identified by its string names. Programs can change the association between the name and the content. Since the content is a value of type `any`, the actual type of the value might change in future. Hence, even though a program runs correctly at a particular time, it may incur run-time type errors in future. On the other hand, a statically typed correct program never goes wrong unless the database schema is invalidated.

### 5.6.3 Linearity checking

Some language systems detect the "serializability" of impure constructs in an expression. A well-known functional programming language is Clean (Achten et al. 1993). Its type checker detects the side effects of expressions and ensures the linearity of the effects not be compromised. Similar linearity analysis is also found in PFL (Sutton and Small 1995). An example of such erroneous expressions is thus

$$(\mathit{include}\ t\ r,\ \mathit{exclude}\ t\ r),$$

where the first subexpression inserts a tuple $t$ to the relation named $r$, and the second one deletes the same tuple. The result, of course, depends on the order of evaluation. PFL ensures that every expression is confluent by making use of *linear typing*. Hence the above expression is *type-incorrect*, and is detected as illegal before execution. Note that linear typing ensures that mutable values are never duplicated nor discarded, and thus results in the language being *confluent*. Even though confluence ensures that every expression is unambiguous, the referential transparency of the language is compromised. In other words, the approach of PFL allows the name-value bindings to be modified in database transactions, making the language referentially opaque.

Lastly, we shall mention LDL (Naqvi and Tsur 1989), a deductive database management system, as an example found in different paradigms. A program in LDL is a Horn clause, whose whose body is a sequence of literals or subgoals. The subgoals in LDL are resolved or executed sequentially in the order defined in the clause *by default*. However, if the language processor can prove that the parallel execution of some subgoals does not affect the semantics of the clause, that is to say, the processor can prove the sufficient condition to ensure the safety, the subgoals may be executed in parallel.

# Chapter 6

# Triggers for Integrity Enforcement

A database model has as an integral part a mechanism to specify integrity constraints of the database state. Persistent programming languages should also have certain primitives to help the designers specify the constraints, but the general purposeness of the languages makes it difficult for a fixed set of built-in primitives to support arbitrary integrity constraints. Thus, instead of facilitating the Haskell language with such extensions, we make use of automatic triggering of functions.

As has been explained, most primitive operations are overloaded, and entity types and persistent roots are associated with `Entity` and `PerRoot`, respectively. This leads to a technique that associates the operators with "hooks" to customize the operations. The idea of using hooks to customize predefined functions prevails in the Lisp community. The same idea is also adopted in Common Lisp Object System (CLOS) (Keene 1989). Thus, the same customizability is inherited by persistent programming languages based on Lisp or CLOS (Fishman et al. 1987; Paepcke 1988; Barbedette 1992). This chapter aims at showing similar customizability is achieved in the context of purely functional database programming, and that the ability to handle multiple versions naturally extends to support the transaction boundary rule execution.

This section first summarizes the concept of active databases, and then proposes a method of incorporating a triggering mechanism into the methodology using the simu-

lation of the type extent model of persistence as an example. The remainder explains how other integrity constraints, i.e., including mutual references, materialized views, and dangling reference exceptions, are supported by the mechanism, and also shows that the transaction-boundary rule-firing is supported.

## 6.1 Active database technology

Active database technology was originally proposed to strengthen the power of integrity enforcement by database management systems (Eswaran and Chamberlain 1975). Since then the concept was generalized to handle more general rules in HiPAC (McCarthy and Dayal 1989), POSTGRES (Stonebraker et al. 1990), Starburst (Widom 1996), and ODE (Agrawal and Gehani 1989). Among these, the HiPAC project proposed Event-Condition-Action (ECA) rules, as a general formalism of active database management functions. An ECA rule, as the name suggests, comprises three parts. The first part specifies the relevant events, such as update of a certain relation, which enable the rule. The second part prescribes the Boolean condition that specifies when to invoke the action given in the third part. In an abstract syntax, the ECA rule can be written as

$$\text{on} < \textit{event} > \text{if} < \textit{condition} > \text{then} < \textit{action} >.$$

Ghandeharizadeh et al. (1996) characterized execution semantics of ECA rules into two aspects: the *coupling modes*, which were identified in the HiPAC project, and the number of state values included in condition checking and action invocation. The coupling modes specify the relative timing among event detection, condition checking, and action execution. The timing is either *immediate*, *deferred*, or *separate*. Immediate coupling means that the second activity follows the first immediately, while deferred coupling means that the second activity is postponed until some later time but still falls within the same transaction. Separate coupling means that a concurrent process is spawned to perform the second activity.



This and the next sections consider only *immediate* and *deferred* coupling modes, since the transaction model considered here does not support concurrent transaction execution. It might be possible to generalize the environment to support concurrent threads as proposed in (Akerholt et al. 1993; Trinder 1995). This paper, however, puts more stress on the applicability of the purely functional programming paradigm to flexible control of database state management. The issues on transaction formalism and architectures are out of its scope.

## 6.2 Simulating the type-extent model of persistency

The reachability model of persistence requires that every piece of data should be reachable from at least one of the persistent roots. This is the source of flexibility, but also the source of complexity of the extent management. In manipulating a database entity, programmers must always take into account the set of related persistent roots. In addition, even though a program works at a particular point, modification of the set of persistent roots would invalidate the program; the newly introduced roots are not necessarily taken into account in the original program design phase.

Remember that a database type is specified by making it an instance of the `Entity` class:

```
> module Parts where
> data Part
>    = Basic     { pName::String,          pCost, pMass::Int,
>                  pUsedBy::[DBRef Part], pSuppliedBy::[DBRef Supplier]}
>    | Composite{ pName::String,          pCost, pMass::Int,
>                  pUsedBy::[DBRef Part], pComposedOf::[(DBRef Part, Int)]}
> instance Entity Part
```

This example specifies no instance methods for `Part`. This implies that the default entity management strategy suffices for this type. If programmers decide to invoke actions

```
> instance Entity Part where
>   afterNew pid part
>     = do PartExt{parts} <- readRootDB
>          writeRootDB PartExt{parts=pid:parts}
```

Figure 6.1: Hook to automatically insert a part into the root

at the entity management operations like *new*, *read*, and *write*, appropriate actions can be given as instance methods. Since it would be dangerous to allow for such direct modification of primitive functions, we design the primitive functions so that they call customizable overloaded functions or *hooks*. By default such companion functions do nothing. If the users define the hooks, they are called automatically according to the database modification.

To illustrate, we define a hook that automatically inserts a newly created `Part` entity into the persistent root associated with `PartExt`. We only have to declare the action as an instance method for `afterNew` as shown in Fig. 6.1. The first argument is the surrogate of the newly created entity, and the second one is the associated value. The specified action gets the root value through `readRootDB`, constructs a new root value, and then updates the root with the new list value. The underlying system automatically calls this hook for application programs. Therefore, the simple expression

```
newDB Basic{pName="part0010", pCost=100, ...}
```

implies that

```
do pid            <- newDB' Basic{pName="part0010", pCost=100, ...}
   PartExt{parts} <- readRootDB
   writeRootDB PartExt{parts=pid:parts}
```

where `newDB'` designates the built-in primitives of the surrogate creation[1].

[1]Of course, the name of the built-in primitive depends on the implementation. We name it just to discriminate it from the `newDB` primitive function.



This style of entity management is very flexible. For instance, we can split the persistent root into two roots: one for `Basic` parts and the other for `Composite` parts. In this case, the persistent roots regarding the `Part` entities may be declared thus:

```
> data BParts = BParts{bparts::[DBRef Part]}
> data CParts = CParts{cparts::[DBRef Part]}
>
> instance PerRoot BParts
> instance PerRoot CParts
```

Now that the root has been split, the `afterNew` method must be modified as follows:

```
> instance Entity Part where
>   afterNew pid Basic{}
>     = do BParts{bparts} <- readRootDB
>          writeRootDB BParts{bparts = pid:bparts}
>   afterNew pid Composite{}
>     = do CParts{bparts} <- readRootDB
>          writeRootDB CParts{bparts = pid:bparts}
```

Splitting the root might affect programs that have already been developed, but the view mechanism solves this problem to some extent. Remember that Fig. 5.3 defines two views for basic and composite parts. A similar definition specifies a view that automatically merges the two lists to construct the full list of stored parts.

Another important difference between the type-extent model of persistence and the reachability model is the ability to allow for explicit deletion of entities. Since simulating this operation is closely related to the mutual references, it will be discussed just below.

## 6.3 Relationship management

Insertion and update of an entity must be properly handled, since relationships between entities may be implemented through mutual references. The above "hooking" technique is also applicable to maintain such mutual references in a terse and declarative way.

Mutual references have been seen in the part-supplier database. A part entity refers to

other parts through the `pUsedBy` and `pComposedOf` fields. Similar relationships also exist between `Part` and `Supplier` values through `pSuppliedBy` and `sSupplies`. Whenever a supplier stops to supply a part, the corresponding `Supplier` entity must be modified accordingly. In addition, the supplied `Part` entity must also be modified to make the mutual references consistent. To support the schema level specification of mutual reference management, two `Entity` class operations, `beforeUpdate` and `afterUpdate` are used as the hooks. These hooks are respectively called before and after `writeDB`.

Figure 6.2 shows the `afterUpdate` hook to maintain the mutual references between parts and suppliers. The hook is called using three arguments: the surrogate, the old value, and the new value. This hook simply checks the "delta" between the old and new values, and modifies the affected parts accordingly. Note that `\\` is the list difference operator, and `delete` deletes the first argument from the second list if it exists.

A similar action may be specified on the `Part` type side. In this case, the `afterUpdate` method for `Part` is defined so that the related supplier entities are modified accordingly. To suppress the infinite calling of hooks, the hook shown above is defined with two *guards*. A guarded right-hand side comprises a condition and an expression written in the following form:

$$f\, pat_1 \cdots pat_n \quad | \quad condition_1 = expression_1$$
$$\vdots$$
$$| \quad condition_m = expression_m.$$

If there are more than one guarded right-hand sides, the first one whose guard is evaluated as being true is selected. In the program shown in Fig. 6.2, the action to maintain the consistency is taken only if the delta between the old and new values is not empty.

The safety of the action is the responsibility of the programmers. While this is not satisfactory, it is theoretically impossible to analyze the programs completely, since the language considered here is general-purpose.

The hooks associated with `writeDB` also make entity deletion easier. For instance,



deleting a part from a database requires us only to clear the fields of the related entities, and to delete the part from the persistent root value:

```
> deletePart pid
>   = do part <- readDB pid
>        case part of
>          Basic{}     ->
>            writeDB pid part{UsedBy=[],pSuppliedBy=[]}
>          Composite{} ->
>            writeDB pid part{pUsedBy=[],pComposedOf=[]}
>        PartExt{parts} <- readRootDB
>        writeRootDB PartExt{parts=delete pid parts}
```

When the `writeDB` function is executed, the `afterUpdate` hook automatically deletes the references to this part in `Part` and `Supplier` entities, so there is no need for the programmers to take into account mutual references associated with the part to be deleted.

## 6.4 Materialized views

A materialized view is implemented by combining a persistent root and related hooks: the root stores the view value, and the hook maintains the value. For example, the materialized views for persistent roots for basic and composite parts are declared through the following instances:

```
> data BParts = BParts {basicParts::[DBRef Part]}
> instance PerRoot BParts
>   initValue _ = BParts{ basicParts = [] }
>
> data CParts = CParts {compositeParts = [DBRef Part]}
> instance PerRoot CParts
>   initValue _ = CParts{ compositeParts = [] }
```

The materialized value management is specified through the triggered routines for the `Entity-Part` instantiation, as shown in Fig. 6.3. The `afterNew` function is the hook that is executed after `newDB` is executed. In this case, it inserts the newly created entity into either of the materialized views according to the kind of the part. The `afterUpdate`

```
> instance Entity Supplier where
>   afterUpdate sid                    -- modified entity's surrogate
>          Supplier{sSupplies=oldParts}  -- oldValue
>          Supplier{sSupplies=newParts}  -- newValue
>
>     | newParts == oldParts             -- if no difference
>     = return ()                        --    do nothing
>
>     | otherwise                        -- else
>     = let added   = newParts \\ oldParts
>           deleted = oldParts \\ newParts
>       in do modifyParts (:)    added
>             modifyParts delete deleted
>   where
>     modifyParts f [] = return ()
>     modifyParts f (p:ps)
>       = do Part{pSuppliedBy=oldValue} <- readDB p
>            let newValue = f sid oldValue
>            writeDB p newValue
>            modifyParts f ps
```

Figure 6.2: The `afterUpdate` method for `Supplier`



```
> instance Entity Part where
>   afterNew pid Basic{}
>      = do BParts{ basicParts  = parts }    <- readRootDB
>           writeRootDB BParts{ basicParts = pid:parts }
>   afterNew pid Composite{}
>      = do CParts{ compositeParts = parts } <- readRootDB
>           writeRootDB CParts{ compositeParts = pid:parts }
>   afterUpdate pid Basic{} Composite{} -- basic to composite
>      = do BParts{ basicParts = parts }     <- readRootDB
>           writeRootDB BParts{ basicParts = delete pid parts }
>           CParts{ compositeParts = parts } <- readRootDB
>           writeRootDB CParts{ compositeParts = pid:parts }
>   afterUpdate pid Composite{} Basic {} -- composite to basic
>      = do BParts{ basicParts = parts }     <- readRootDB
>           writeRootDB BParts{ basicParts = pid:parts }
>           CParts{ compositeParts = parts } <- readRootDB
>           writeRootDB CParts{ compositeParts = delete pid parts }
>   afterUpdate pid _ _
>     = return ()
```

Figure 6.3: Materialization management routines for the basic and composite parts views.

function handles the case where the kind of a part is changed from basic to composite or inversely. When the kind is not changed by the part entity update, no action is taken.

The usage of materialized views is not different from that of ordinary roots. For example, selecting expensive basic parts costing more that 100 dollars, can be performed simply by traversing the `BParts` persistent roots:

```
> expensiveBasicParts
>   = do BParts{basicParts} <- readRootDB
>        parts              <- mapM readDB basicParts
>        return [ pName | Basic{pName, pCost} <- parts, pCost > 100 ]
```

Note that this code does not differ much from a code which directly selects required values from the list of all the parts. Since the scan is performed only on the basic values, however, the run-time performance should be better especially if the ratio of basic parts to all parts is low.

## 6.5 Exception handling

Even though the reachability model of persistence is adopted, an invalid-reference exception occurs when a newly created entity surrogate is looked up in an *older* database state. The following trivial example exhibits such a case:

```
> staleDereference name cost mass
>   = do db <- getDB
>        s  <- newDB Basic{ pName = name, pCost = cost, pMass = mass,
>                           pSuppliedBy = [], pUsedBy = [] }
>        return (readRef(db) sid)
```

While the newly created surrogate is valid only after the `newDB`, the first argument of `readRef` is created before the operation. What is worse, the call-by-need semantics reveals this erroneous situation only when the value is required, not when the value is defined.

Facilitating hooks for database operations can be applied to this case. So we make



the exception handler one of the class operators like this:

```
> class Entity a where
>     whenDangling :: Database -> DBRef a -> a
>     whenDangling _ _ = error "dangling reference"
```

In this declaration, `whenDangling` is declared with its default method. This method is used when the instances do not declare their own methods explicitly. Recall also that dereference is performed by the `readRef` operator. Whenever it detects an invalid surrogate, it applies `whenDangling` to the current database and the given surrogate. For example, let `db` denote a value of type `Database`, and `v` denote a surrogate that is invalid in `db`. Then the following equivalence holds:

`readRef db v` $\equiv$ `whenDangling db v` $\equiv$ `error "dangling reference"`.

When some default value exists, users may specify the value in instance declarations like this:

```
> instance Entity Part where
>     whenDangling _ _ = Basic "B000" 0 0 []
```

In this case, if a surrogate `v` is invalid in a database `db`, the following equivalence holds:

`readRef db v` $\equiv$ `whenDangling db v` $\equiv$ `Basic "B000" 0 0 []`.

# Chapter 7

# Generalization to Active Rules

The previous chapter introduced several hooks so that the behaviors of the primitive database operators are customizable. In terms of active databases, events on database versions are their modification performed by `newDB` or `writeDB`. And the condition and action parts are specified in the associated hooks of the primitives. All the examples described for the integrity enforcement so far are *immediate-immediate* coupling. They only see the current state of the database or the changes of values before and after updates.

This chapter generalizes this idea.The capability to handle multiple versions plays the key role, and the newly introduced database state component, a job queue, makes rule executions more flexible.

## 7.1 Transaction boundary rule execution

The strategy adopted here introduces a queue of jobs. A job comprises the condition and action parts. The coding strategy of the job gives the coupling mode of condition checking and action execution. If the job checks the condition and then takes a certain action immediately, the condition-action coupling mode is "immediate". On the other hand, the job may enqueue the action again as a non-conditional job. In this case, the coupling mode is "deferred". The queue is organized according to a certain queuing discipline. Under the simplest discipline, the queue may be organized as a first-come,

first-served structure. A more flexible discipline may associate precedence values with the jobs in the queue. To support both flexibility and simplicity, we use a queue sorted by precedence values and then by the order of enqueuing.

The queue is controlled by the following primitive operation:

```
> enqueueDB :: Int -> (Database -> DB ()) -> DB ()
```

The first argument specifies the precedence of the job, while the second one specifies the job itself. Note that the action is of type `Database -> DB ()` instead of `DB ()`. This reflects the semantics of the queue execution. The proposed approach basically follows the *transaction boundary rule firing*, under which the enqueued jobs are executed at the end of the surrounding transaction. The relevant database state values may be shown like this

$$db_{orig}, \cdots, db_{prop}, \cdots, db_{cur},$$

where $db_{orig}$ is the original database state at the transaction start time, $db_{prop}$ is the proposed state when the commit procedure starts, and $db_{cur}$ is the current database state. In the examples in the rest of this section, these values are usually bound to `db0`, `db1` and `db2`, respectively.

As an example, consider a situation where the type-extent model of persistency is simulated through the `afterNew` class method. The implementation following immediate-immediate coupling has already been shown in Fig. 6.1. On the other hand, if the action is expected not to be taken immediately, he following rewriting of the code is enough.

```
> instance Entity Part where
>   afterNew pid part
>     = enqueueDB 0 (\db1 ->
>         do PartExt{parts} <- readRootDB
>            writeRootDB PartExt{parts=pid:parts} )
```

After the surrounding transaction starts the commit procedure, the enqueued actions are executed. This example shows the method of writing a non-conditional deferred action.



In this coding, the root update can be observed only after the current transaction has been committed.

The enqueued job may refer to related values to see if its action should be taken. To illustrate, let us consider the case where a supplier who supplies no parts should be deleted. This cannot be simply executed by `afterUpdate`. Even if a supplier supplies no parts at a certain point of a transaction, the supplier value may be modified so that it supplies some other parts in the same transaction. A possible solution is to enqueue an appropriate job whenever a relationship between a supplier and a part is modified. The typical specification may look like this:

```
> instance Entity Supplier where
>   afterUpdate sid oldVal newVal
>     = enqueueDB 0 (aJob sid)
>   where
>     aJob sid db1
>       = do Supplier{sSupplies} <- readDB sid
>            when (sSupplies == []) ( do
>               SuppExt{suppliers=supps} <- readRootDB
>               let supps' = delete sid supps
>               writeRootDB SuppExt{supplies=supps'} )
```

where `when` is a library function defined as

```
> when p a = if p then a else return ()
```

Even though it is not syntactically clear, the condition and the action part of the ECA rule are encoded in this job.

So far the several parameters supplied to the jobs or the methods have not been fully utilized. The following example makes use of the original state of the transaction: when the cost of a basic part is reduced by more than 20%, stops the program with an error message. The program looks like this:

```
> instance Entity Part where
>   afterUpdate pid oldVal newVal
>     = enqueueDB 0 (aJob pid)
>     where
>       pred pid Basic{pCost=cost1} Basic{pCost=cost2}
>         = fromInt cost1 / fromInt cost2 < 0.8
>       pred pid oldVal             newVal
>         = False
>       aJob pid db1
>         = do db0 <- getOrigDB
>              when (pred pid (readRef(db0) pid) (readRef(db1) pid))
>                error ("Part " ++ pName (readRef(db0) pid)
>                    ++ ": cost reduction error")
```

In this program, the original state `db0` and the proposed state `db1` are used to compute the difference between the original and proposed costs. If `db1` were replaced with the current database retrived by `getDB`, the difference between the original and current values would be computed.

Queued jobs are executed at the commit time until the job queue becomes empty. More specifically, the state transition sequence in a transaction looks like this

$$db_{orig}, \cdots, db_{prop_1}, db_{prop_2}, \cdots, db_{cur},$$

where $db_{prop_1}$ is the state at the commit time, and $db_{prop_2}$ is the state after the rules in the queue at the commit time have been executed, and so on. Remember that the type of `enqueueDB` is

```
> enqueueDB :: Int -> (Database -> DB  ()) -> DB ()
```

The $db_{prop_i}$ is passed to all the jobs in the $i$th phase as the argument value supplied to the second argument.



## 7.2 Fixed point iteration

Instead of the semantics of queued job termination, the fixed point semantics could be adopted. In this semantics, the contents of the job queue at the commit time are treated as a single rule group. The rules in this group is repeatedly executed until the database state reaches a fixed point. The fixed point semantics exhibits certain similarities to the forward-chaining of the production rules. Some applications, e.g., decision support systems, may find the semantics more useful than the the simple queuing semantics.

The fixed point semantics of queued job termination may be simulated as follows. Let a job is enqueued in the phase $i$ of the enqueued job execution. Then, to support the fixed point semantics, the rules must be designed so that

1. The queued job has an parameter to access the previously proposed database state, $db_{prop_{i-1}}$;
2. The job's pre-condition checks if $db_{prop_{i-1}}$ passed via the parameter and the current proposed state, $db_{prop_i}$, are same, and only if they differs, enter into its action; and
3. The action part should enqueue itself with the $db_{prop_i}$ passed via the parameter.

Therefore, adding a simple job following the fixed point semantics relative to the proposed state can be achieved by the function shown in Figure 7.1 (a). In the function, `Nothing` and `Just` are the type constructors of the `Maybe` type. The purpose of using `Maybe` is to distinguish the first job firing and others. The pre-condition, `prePred`, checks if the previous proposed database state, if any, differs from the current proposed state. Only if they differs, `aJob` executes the given condition checking and the given action, followed by the queuing of the job again. Note that the rule specification shown above follows the fixed point semantics only if all of the queued rules follow this semantics.

The fixed point iteration may be defined relative to either the proposed state or the current state. The fixed point semantics implemented above uses the former iteration.

```
> addFixedPointRule ::
>      Int -> (Database -> Database -> Database -> Bool)
>          -> (Database -> DB ())
>          -> DB ()
> addFixedPointRule prec cond action
>  = let prePred Nothing     db1 = True
>        prePred (Just db1') db1 = db1' /= db1
>
>        aJob db0 prevDb1 db1
>          | prePred prevDb1 db1 = do db2 <- getDB
>                                     when (cond db0 db1 db2) (action db1)
>                                     enqueueDB prec (aJob db0 (Just db1))
>          | otherwise           = return ()
>
>    in do db0 <- getOrigDB
>          enqueueDB prec (aJob db0 Nothing)
```

(a) The function to add a rule following the fixed point semantics.

```
> addRuleIteration ::
>    Int
>     -> [ (Database -> Database -> Database -> Bool, Database -> DB ()) ]
>     -> DB ()
> addRuleIteration prec ruleList
>  = let mkJobList db0 db1 db2
>          = [ when (cond db0 db1 db2) (action db1)
>              | (cond, action) <- ruleList]
>
>        aJob db0 db1
>          = do db2  <- getDB
>               sequence (mkJobList db0 db1 db2)
>               db2' <- getDB
>               when (db2 /= db2') (aJob db0 db1)
>
>    in do db0 <- getOrigDB
>          enqueueDB prec aJob
```

(b) The function to add a set of rules which iterate until state reaches a fixed point.

Figure 7.1: Two utility functions to manipulate rules following the fixed point semantics.



On the other hand, the latter iteration is implemented by combining a set of jobs into a single fixed point iteration job. This can be easily achieved by the flexibility of functional programming as shown in Figure 7.1 (b). Notice that the rule execution is controlled using the current state,db2, instead of the proposed state, db1.

## 7.3 Extension level rule specification

In so far, the active rule execution is embedded in the class methods for Entity instances. In other words, these active rules are specified at the scheme level, instead of at the database extension level. This approach has an advantage that view materialization and integrity constraint enforcement are specified declaratively in a database schema. In contrast with this rule specification, the extension level specification of database rules is more flexible and is easier to modify so that they reflects more frequently changing criteria. For example, a decision support system on a stock database may trigger a certain rule when a particular stock price changes. This activation reflects the interest of an enterprise at a particular point, and may be changed more frequently than the database schema.

This section describes the method to support this sort of rule specification through persistency roots storing type-specific actions. The database action values are represented by the DBRule data type shown in Figure 7.2. The four data constructors are respectively to specify rules for the *before/after new* and *before/after update*. The rest of the work is to specify persistency roots to store the set of active rules as shown in Figure 7.3. The afterNew method automatically enters the newly created rule value into the corresponding list. In this example, rule values are stored in the list instead of the rule surrogates. Hence, the role of making the DBRule type an instance of the Entity class is to lessen the burden of rule programmers.

The primitive database operator such as newDB retrieves this list of rule groups and apply them in the order of specification. Let $v$ be the value of a newly created entity,

```
type OrigDB = Database
type  CurDB = Database

data Entity a => DBRule a
   = beforeNew
        ruleName                :: String,
        ruleIsActive            :: Bool,
        rulePriority            :: Int,
        ruleBeforeNewCondition :: OrigDB -> a -> CurDB -> Bool,
        ruleBeforeNewAction     :: a -> DB () }
   | AfterNew {
        ruleName                :: String,
        ruleIsActive            :: Bool,
        rulePriority            :: Int,
        ruleAfterNewCondition  :: OrigDB -> DBRef a -> a -> CurDB -> Bool,
        ruleAfterNewAction      :: DBRef a -> a -> DB () }
   | beforeUpdate {
        ruleName             :: String,
        ruleIsActive         :: Bool,
        rulePriority         :: Int,
        ruleUpdateCondition :: OrigDB -> DBRef a -> a -> a -> CurDB -> Bool,
        ruleUpdateAction     :: DBRef a -> a -> a -> DB () }
   | AfterUpdate {
        ruleName             :: String,
        ruleIsActive         :: Bool,
        rulePriority         :: Int,
        ruleUpdateCondition :: OrigDB -> DBRef a -> a -> a -> CurDB -> Bool,
        ruleUpdateAction     :: DBRef a -> a -> a -> DB () }
```

Figure 7.2: Data types to specify rules



```
data Entity a => DBRuleSet a = DBRuleSet [DBRule a]

instance Entity a => PerRoot [DBRule a] where
   initValue _ = DBRuleSet []

instance Entity a => Entity (DBRule a) where
   afterNew s v = do DBRuleSet rs <- readRootDB
                     writeRootDB (DBRuleSet (v:rs))
```

Figure 7.3: Root to store rule values

— before creation.
*retrieve the* `BeforeNew` *rule set for this type*
*execute active rules in the set with negative priority values*
*execute* `beforeNew` *method for this type*
*execute active rules in the set with non-negative priority values*
— create the surrogate.
*create a new entity with value v*
— after creation.
*retrieve the* `AfterNew` *rule set for this type*
*execute active rules in the set with negative priority values*
*execute* `afterNew` *method for this type*
*execute active rules in the set with non-negative priority values*

Figure 7.4: Internal steps to execute `newDB`

```
> addRule =
>  let -- condition part
>      pred db0 pid Basic{pCost=cost1} Basic{pCost=cost2} db2
>        = fromInt cost2 / fromInt cost1 < 0.8
>      pred db0 pid oldVal             newVal             db2
>        = False
>
>      -- print the warning message
>      printMsg pid oldVal newVal
>        =  ... generate warning message ...
>
>      -- rule value
>      aRule =
>        AfterUpdate {
>         ruleName            = "basic-part-rule1",
>         ruleIsActive        = True,
>         rulePriority        = 0,
>         ruleUpdateCondition = pred
>         ruleUpdateAction    = printMsg }
>  in newDB aRule
```

Figure 7.5: Rule to check the reduction of basic part costs.

then the primitive operator, `newDB` is internally executed following the the steps shown in Figure 7.4. `beforeUpdate` and `afterUpdate` are treated in the similar manner.

To illustrate, consider that the action that prints a warning message whenever the cost of a basic part is reduced more than 20%. The rule specification is shown in Figure 7.5. The rule value is automatically inserted into the corresponding rule list. More complex conditions and actions may be specified in the same manner.

Job queuing described in the last section may be mixed with the extension level rule specification to make use of the transaction boundary execution semantics. The rule shown in 7.5 is modified as such by the following three steps: (1) relax the condition; (2) compose the original condition and the action into a single action; and (3) specify the queuing action as the `ruleUpdAction` value. The modified code is shown in Figure 7.6.



```
> addBoundaryRule =
>  let -- condition part of the action
>      pred pid Basic{pCost=cost1} Basic{pCost=cost2}
>        = fromInt cost2 / fromInt cost1 < 0.8
>      pred pid oldVal             newVal
>        = False
>
>      -- print the warning message
>      printMsg pid oldVal newVal
>        =  ... generate warning message ...
>
>      -- queued job itself
>      aJob pid oldVal newVal
>           | pred pid oldVal newVal = printMsg pid oldVal newVal
>           | otherwise              = return ()
>
>      -- action
>      action pid oldVal newVal
>        = enqueueDB 0 (\db1 -> aJob pid oldVal newVal)
>
>      -- condition to queue the job
>      qPred db0 pid Basic{}  newValue db2 = True
>      qPred db0 pid oldValue newValue db2 = False
>
>      -- rule to enqueue the job.
>      aRule =
>        AfterUpdate {
>         ruleName            = "basic-part-rule2",
>         ruleIsActive        = True,
>         rulePriority        = 0,
>         ruleUpdateCondition = qPred,
>         ruleUpdateAction    = action }
>  in newDB aRule
```

Figure 7.6: Rule to check basic part cost reduction. (The transaction boundary semantics is adopted.)

The notable change is made in the `ruleUpdateAction` value. Whilest the original value was the action to print the warning message, the new action is to enqueue the jobs which checks if the reduction ratio is 20% or more, and if so, prints the warning message.

If the action must be taken only when the reduction ratio of the value in the proposed state to the one just before the modification, then the `action` has to be changed as follows:

```
>      action pid oldVal newVal
>        = enqueueDB 0 (\db1 ->
>              aJob pid oldVal (readRef(db1) pid))
```

where `aJob` is supplied with the value retrieved from the proposed state, `db1`, via the `readRef` operation.

Since an extension-level rule is just a stored value, it can be deleted like this

```
>    do DBRuleSet rs <- (readRootDB :: DB (DBRuleSet Part))
>       let newRules = [ r | r <- rs, ruleName r /= "basic-part-rule1" ]
>       writeRootDB (DBRuleSet newRules)
```

## 7.4 Further generalization and performance issues

We have already explained two functions, `beforeUpdate` and `afterUpdate`, which respectively handle the job triggering based on the difference between the entity values before and after a certain update action. It is not difficult to generalize these functions so that the triggering is based on the difference between the database state values before and after the update operation. Consider another class operator of `Entity`, say `checkUpdate`. The new class operator would be typed as follows:

```
>  checkUpdate :: DBRef a -> Database -> Database -> DB ()
```

The first argument is the updatee, and the second and the third arguments are respectively the database versions before and after the `writeDB` operation. Although this operator is



not difficult to implement, it would pose certain amount of performance degradation; since every update would lock the database state before the action, every update action generates a new version. This issue is easy to see by considering state transitions explicitly. According to the formalization explained in 5.5, the operation denotes an expression like this

$$\begin{array}{l}\texttt{getDB >>=}\ (\lambda\ d.\\ \texttt{writeDB}\ i\ v\ \texttt{>>=}\ (\lambda\ ().\\ \texttt{getDB >>=}\ (\lambda\ d'.\\ \texttt{checkUpdate}\ i\ d\ d'))).\end{array}$$

Expanding `getDB` and `>>=` with slight simplification gives the following equivalent expression:

$$\begin{array}{l}\lambda\ (x, g).\ \textbf{let}\ ((), (x'', g'')) = \texttt{writeDB}\ i\ v\ (lock\ x, g)\\ \qquad\textbf{in}\ \texttt{checkUpdate}\ (lock\, x, g)\ (lock\, x'', g)\ (x'', g'').\end{array}$$

Lastly, expanding `writeDB` gives

$$\begin{array}{l}\lambda\ (x, g).\ \texttt{checkUpdate}\ (lock\, x, g)\\ \qquad\qquad (lock\, (update\ i\ v\ (dup'\ (lock\ x))), g)\\ \qquad\qquad (update\ i\ v\ (dup'\ (lock\ x)), g).\end{array}$$

Since $dup'\ (lock\ x)$ is equivalent to $clear\ (dup\ x)$, it generates a *new* database state. By explicitly name it as $x'$, we now have the following expression:

$$\begin{array}{l}\lambda\ (x, g).\ \textbf{let}\ x' = clear\ (dup\ x)\\ \qquad\textbf{in}\ \texttt{checkUpdate}\ (lock\, x, g)\\ \qquad\qquad (lock\, (update\ i\ v\ x'), g)\\ \qquad\qquad (update\ i\ v\ x'), g).\end{array}$$

As indicated in $x'$, every update operation must duplicate its current database state.

In contrast with this generalization, the usage of $db_{orig}$, $db_{prop}$ and $db_{cur}$ does not incur the problem pointed out just above. $db_{orig}$ is always locked at the start time of a transaction, and this version is kept intact during the transaction execution. Hence the passing the $db_{orig}$ does not incur additional cost. Although $db_{cur}$ is locked explicitly when a condition value of extension level rule is evaluated, all the heap-allocated data generated in the evaluation process become garbage immediately after the boolean value

of the condition is calculated. Therefore, the internal implementation may unlock the current state immediately after the condition evaluation, suppressing unnecessary state duplication. On the other hand, locking $db_{prop}$ incurs an additional cost. Because actions that are taken after $db_{prop}$ is locked may update the database state, duplication of $db_{prop}$ may occur. This duplication, however, at most once for each transaction. $db_{prop}$ is shared by all the performed actions after the state is locked.

## 7.5 Advantages and further research issues

This section concludes these two chapters by discussing the advantages and the disadvantages of the proposed active database mechanism. All of the features are achieved by inherent flexibility and safety of the proposed method.

**Static typing and referential transparency.** Rules are triggered through the overloaded functions associated with the database entity types. The type system ensures that an appropriate database rules are triggered according to the entity type, and the programs, which may include seemingly side-effecting actions, are referentially transparent and statically typed.

**Mixture of schema level and extension level rules.** In addition to the rules specified in database schemas, rules can be treated as stored database values. Scheme level rules are suitable to declaratively specify integrity constraints, and view materialization, while rules at the extension level support rules that change more frequently than database schemas.

**Handling dangling reference exception.** An inherent problem in handling multiple database versions is the exception caused by dangling references. An attempt to dereference a dangling reference of an entity type is also treated as an event occurring in a database state, and controlled by the schema designers.



**Job queuing for flexible coupling control.** Job queuing is adopted to support transaction boundary semantics of rule firing, and the deferred coupling of the event-condition and condition-action coupling modes. Besides the standard semantics, i.e., the rule iteration until the job queue becomes empty, the fixed point semantics can be implemented. This is achieved by the flexibility of function programming and the ability to manipulate multiple versions. The fixed point iteration relative to the current state can be implemented without difficulty through the same features of the proposed database manipulation interface.

The topics requiring further investigation include the following:

**Declarative rule specification.** The notable disadvantage is that rules are not declarative in contrast to those defined in rule sublanguages. A dedicated rule sublanguage might allow programmers to write a rule as follows:

```
define rule basic-part-cost-rule1
update to part.cost
if  kind = 'basic' and
    part.cost / old part.cost < 0.8
then ... print the warning message ...
```

This is much simpler than the rule shown in Figure 7.5. We could design a simple rule language and the preprocessor which translates a rule specification into a database action to add corresponding rule. For example the rule in Figure 7.5 might be specified in a virtual rule language as follows:

```
define rule basic-part-cost-rule1
on after update to part with priority 0
immediately if
    fromInt (pCost newVal) / fromInt (pCost oldVal) < 0.8
immediately do
    ... print warning message ...
```

Even though this virtual rule sublanguage would simplify the rule specification for simpler situations, the flexibility of the proposed method might be lost.

**Rule optiomization and conflict resolution.** Analysis of the rules is theoretically impossible, since Haskell is a computationally complete programming language. This raises two issues. Firstly it is difficult to facilitate the underlying system with a low-level optimization method like *Rete* (Forgy 1982). Secondly the conflict between database modification operations is not detectable as in (Ghandeharizadeh et al. 1996).

**Composite events and non-database events.** Only atomic database upates are treated as events. Generalized events include, for example, composite events with variables, and non-database events such as temporal events (Motakis and Zaniolo 1997). Simply tracing multiple database update events might be implemented by recording the update history and by checking triggering conditions on it with certain amount of additional execution cost.

Non-database events are difficult to incorporate properly in the purely functional processing. Although state transformers support the database state concept and direct state update, this does not mean that general side-effecting operations are permitted. Haskell language does not support general-purpose event-based I/O mechanism. An event-driven I/O mechanism in Clean (Achten et al. 1993) might be integrated with the monadic I/O mechanism of Haskell to extend the I/O functionalities, but the issue is out of scope of this dissertation.

# Chapter 8

# Implementation Issues

All the objects allocated in a persistent programming environment are qualified to survive from one user session to another. The simplest implementation method of supporting this persistency is to dump the memory image of a language processor at the end of one session, and reload the image at the next start-up time. This technique is often used in volatile programming environments to save the cost of loading library functions. From the view point of persistent programming, however, this method has drawbacks; the environment does not scale to a multi-user one, and the startup- and commit-procedures become more time-consuming according to growth of the database population.

Therefore, the current prototype has been built by modifying a Haskell interpreter, Hugs 1.3[1], so that it constructs cells on arrays stored in a persistent heap system, with Texas Persistent Store (Singhal et al. 1992)[2] that supports C++-based persistent programming environment[3]. Although this layered architecture does not necessarily allow for multi-user concurrent access to the persistent heap, the atomicity of user sessions can

[1]This is a partial implementation of Haskell 1.3 by Mark P. Jones, and is short for *Haskell Users Gofer System*. The newest revision at the time of writing is Hugs 1.4 that supports the newer Haskell specification 1.4, and is the joint work by Mark P. Jones at the University of Nottingham and Yale Haskell Group. The distinguishable difference between these two Hugs softwares are support of the Haskell module system, and compliance to Haskell Standard Library 1.4. The internal structures of these two interpreters do not differ so much, though.

[2]The current revision is 0.5.

[3]Since Hugs 1.3 is written in C, the internal functions have been modified so that they are C++-compliant.

be guaranteed by the underlying storage system.

The language processor allocates cells dynamically, and reclaimed the space occupied by unnecessary or *dead* cells automatically. These dead cells are often called *garbage*, so the reclamation procedure is called a *garbage collector*. Hugs uses a simple *mark-and-sweep* garbage collector to support a heap of fixed-length cells, and also uses a secondary *two-space* garbage collector to support variable-length cells. The mark-and-sweep garbage collector scans all the fixed-length cells reachable from prescribed C-variables, calling stacks, and internal bookkeeping tables to determine which cells are live (*maring phase*), and then collects the space occupied by the garbage to make a list of free cells (*sweep phase*). The two-space garbage collector collaborates in the marking phase, and the live variable-length cells are *scavenged* before the sweep phase.

It has been pointed out that the performance of the mark-and-sweep collector is not good (Wilson 1992), but the performance of the garbage collector itself is out of the scope. This dissertation, on the other hand, focuses on the method of supporting the version-surrogate space inhabited by the database entities. This issue is important because of the following points.

- The entity-dereference procedure is a function from versions and surrogates to values, so it includes searching on these "key" values. The organization of the space affects the performance of the entity-dereference procedure.

  There are two straightforward methods of implementing this version-surrogate space. The first method slices the space by versions, and the the other slices it by the surrogates. The former method associates every version with its set of surrogate-value bindings in the version. The other method associates every object with its history of values. In general, the latter methods saves space, since the internal state of an object is independent of others. The current prototype uses this surrogate-slicing.



- The system requires a bookkeeping mechanism which records the relationships among versions and surrogate-value bindings. This bookkeeping may result in phantom live cells, which are pointed to only from the bookkeeper and so should be reclaimed to improve the access performance. This phenomenon is called *space leak* in general.

  To avoid the space leak problem caused by the bookkeeping mechanism, the current prototype has modified the garbage collector conservatively; if a version cell is pointed to only from its descendents, then the cell is considered to be dead and the space is reclaimed. In this reclamation phase, the version history structure is reorganized so that the death of the version is considered in order to improve the performance of dereferencing.

Database operations are implemented in Haskell using the newly introduced built-in functions that control the persistent heap and versions. This dissertation does not enter into the details of the Haskell part of the prototype, since the database monad is a kind of strict state-transformer monads described by Peyton Jones and Wadler (1993). The exception is the persistency-root identification method. This requires the modification of the language processor, so we will mention the technique briefly.

The remainder of this chapter begins with a sample user interaction with the current prototype system, and then focuses on the implementation techniques of persistent roots, and versions. The topics specific to Hugs and Texas Persistent Store are covered in Appendix A.

## 8.1 A sample interaction with the prototype

Fig. 8.2 shows the start-up message of the user-interaction with the current prototype; after setting the `PHUGS` environment variable to the directory where the persistent heap

```
pfp4% setenv PHUGS /home/ichikawa/lang/hugs1.3/pstore/parts
pfp4% pHugs
Hugs, The Haskell User's Gofer System Version 1.3beta
Copyright (c) Mark P Jones, The University of Nottingham, 1994-1996.

 Hacked: g++ compliant, reentrant, and so on.. see storage.h
TABALLOC_ps          gregs : 0x08127008 (135426056)
Already initialized.
TABALLOC_ps        heapFst : 0x0815d008 (135647240)
TABALLOC_ps        heapSnd : 0x081e2008 (136192008)
TABALLOC_ps      flatspace : 0x08266008 (136732680)
TABALLOC_ps           text : 0x08129008 (135434248)
TABALLOC_ps       textHash : 0x08141008 (135532552)
TABALLOC_ps      tabSyntax : 0x08128008 (135430152)
TABALLOC_ps      tyconHash : 0x08147008 (135557128)
TABALLOC_ps       tabTycon : 0x08148008 (135561224)
TABALLOC_ps       nameHash : 0x08147608 (135558664)
TABALLOC_ps        tabName : 0x08332008 (137568264)
TABALLOC_ps       tabClass : 0x0811d808 (135387144)
TABALLOC_ps        tabInst : 0x0814a008 (135569416)
TABALLOC_ca      cellStack : 0x0838e000 (137945088)
TABALLOC_ps        modules : 0x0814e008 (135585800)
TABALLOC_ca        handles : 0x083a6000 (138043392)
TABALLOC_ps     scriptName : 0x08375008 (137842696)
TABALLOC_ps     lastChange : 0x08126008 (135421960)
TABALLOC_ps      postponed : 0x08126188 (135422344)
```

Figure 8.1: Opening message



```
Hugs session for:
-[  0] Prelude.hs
-[  1] /home/ichikawa/lang/hugs1.3/lib/List.hs
-[  2] /home/ichikawa/lang/hugs1.3/lib/IO.hs
-[  3] /home/ichikawa/lang/hugs1.3/libhugs/IORef.hs
-[  3] /home/ichikawa/lang/hugs1.3/libhugs/IORef.hs
-[  4] /home/ichikawa/lang/hugs1.3/lib/Monad.hs
-[  5] /home/ichikawa/lang/hugs1.3/libhugs/ST.hs
-[  6] /home/ichikawa/lang/hugs1.3/lib/Array.hs
-[  7] /home/ichikawa/lang/hugs1.3/libhugs/STArray.hs
-[  8] DBState.hs
-[  9] Part.hs
 [ 10] query.hs
 [ 11] Restore.hs
Type :? for help
?
```

Figure 8.2: Starting up a user session

image is stored, the `pHugs` command starts a session[4]. The opening message also includes information concerning the persistent and the volatile storage allocation. The message is followed by the lines indicating the names of the loaded scripts as shown in Fig. 8.2, where "?" is the prompt of the system. Some script information lines start with "-". This indicates that the script is "locked", so that these scripts are not reloaded accidentally. Among others, reloading `DBState.hs` clears the persistent heap, and reloading `Part.hs` that defines the schema invalidates database objects. If anther database types are to be added to the environment, appending the defining script suffices.

A user interacts with the system by typing expressions. For example, a query that retrieves all the basic parts which cost more than $100 can be formulated as follows:[5]

[4]The original system executes the interpreter with the command `Hugs`.

[5]The current interface does not support multi-line inputs, but the expression shown here is split into multiple lines to improve the readability. The expression, however, can be typed in a single line without modification.

```
? task2 ()
("B007",157)
("B005",156)
("B004",119)
("B003",111)
("B002",115)
("B001",103)
("B000",168)

? doJob task3c
("C041",15050592,5490717)
("C040",26499458919550,8445022145585)
("C039",2525088003253738,793534790155882)
... omitted ...
("C011",83135,24654)
("C010",45431,18111)
("C009",54434,19890)
```

Figure 8.3: Sample execution

```
? do { bNames <- transaction (
                   do { PartExt{ps}<-readRootDB
                        accumulate (map readDB ps) } );
       print basicParts }

[Basic{pName="B001",pCost=40,pSuppliedBy=[]}, ....
```

The internal do expression is of type `DB [Parts]`, and is taken to the I/O world by the `transaction` function. This action is finally combined with the `print` function. The result of this execution is of type `IO ()`.

It is onerous to type more complicated expressions interactively. In general, a user formulates tasks which (s)he wants to perform as functions in a script with various parameters. The `query.hs` script, which is the tenth script in the environment shown in Fig. 8.2, defines functions implementing tasks described in Chapters 3 and 5. Fig. 8.3 illustrates the execution of the tasks 2 and 3; `task2` queries the names of expensive basic



parts, and `task3c` computes recursively the total mass and cost of all the composite parts. Notice that `task2` is applied to `()`. This is required because of the property of *constant applicative forms* (CAFs)[6]. Otherwise the result of computation would be cached and never be recomputed even after the database state is modified. `task3c` is supplied to the `doJob` utility function, which retrieves the current database state, passes it to the given function, and prints the results.

While these examples are read-only transaction, user's expressions may modify the database state in general. The current prototype has been equipped with three meta-commands to control user's session; the `:commit` and `:abort` meta-commands end the current user's interaction, respectively, committing and undoing the modification during the session, and the `:check` commits the modification without terminating the interaction. Note that top-level nested transactions are not supported just because Texas Persistent Store does not support it.

## 8.2 Implementing persistent roots

The database state includes a set of persistent roots. Every root location is designated by its type, and the database state maps a persistent root type to the corresponding root value. This section describes the current implementation of the persistent roots, and the database state.

Designation of a persistent root by its type requires the type representation of an expression to be computed at run-time. The task could be considered as *reflection* (Kirby 1992), or *dynamic typing* (Dearle et al. 1989; Abadi et al. 1991; Peterson 1994). Among these, Peterson (1994) pointed out that the type information is available at run-time using either the class mechanism to generate type annotation, or the typing routine executed at run-time. The latter approach has been implemented in *Yale Haskell Compiler 2.2* (Yale

[6]For more detail, see A.3.1.

Haskell Group 1994). This dynamic typing technique, however, requires modification of the language syntax, type checker, byte-code sequences, and dictionary management, and may raise run-time typing errors.

As described in Chapter 4, we only have to generate type annotations for ground types, so the former method suffices, though this method has a drawback, i.e., the unexpected appearance of classes in inferred types. The rest of this section describes the method of generating type annotations at run-time by utilizing the class mechanism, the modification of the language processor to lessen the user's burden, and the database state structure which maps the type annotation of a root to its value.

### 8.2.1 Generating type annotation

Type annotations are generated through the following type class and function:

```
> class Typeable a where
>     typeSig :: TSignature a
>
> typeof :: Typeable a => a -> TSignature a
```

where "`TSignature` $a$" is an abstract type to record the type annotation of $a$. The function `typeof` computes the type annotation of a value, provided that the type is an instance of `Typeable`. The details of `TSignature` are not important here. Instead, we only point out that a value of this type can be constructively generated; for a type $t\ a_1\ a_2\ \cdots\ a_n$, if the type annotation of $a_i$ $(1 \leq i \leq n)$ are given, and the constructor annotation of $t$ is also known, then the type annotation of $t\ a_1\ a_2\ \cdots\ a_n$ is computable at run-time.

The annotation for a type constructor of arity $i$ is given by the class `TypeableCon`$i$ declared partly as follows:



```
> class TypeableCon0 m where
>     mkTSig0 :: TSignature m
>
> class TypeableCon1 m where
>     mkTSig1 :: Typeable a =>
>          TSignature a -> TSignature (m a)
>
> class TypeableCon2 m where
>     mkTSig2 :: (Typeable a, Typeable b) =>
>          TSignature a -> TSignature b -> TSignature (m a b)
```

The argument(s) of the `mkTSig`$i$ is(are) the type annotation(s) of the constructor arguments. To illustrate, we show the instance declarations for the list type constructor, `[]`:[7]

```
> instance TypeableCon1 [] where
>   mkTSig1 tsig = f Nothing
>     where f :: Maybe [a] -> TSignature [a]
>           f d = TSig (SigAp (SigCon "Prelude" "[]") (sigval tsig))
>
> instance Typeable a => Typeable [a] where
>     typeSig = mkTSig1 typeSig
```

Note that `typeSig` is recursively defined, and that the type of tis method in this case is

```
typeSig :: Typeable a => TSignatrue a -> TSignature [a].
```

The one in the right-hand side computes the annotation of the argument type, and `mkTSig1` uses this annotation to compute the overall annotation of the list type. This typing relationship between the argument type and the overall type is ensured by the type of this method.

The following interaction exemplifies the usage of the `typeof` operator:[8]

```
? tsigToStr (typeof "abcd")

"Prelude.[]{Unknown.Char}"
```

[7] The internal detail is simpler since we only have to record the module where the constructor is defined and the name of the constructor. We omit it to avoid the explanation of the internal details.

[8] The constructor names include the "Unknown" prefix. This is just because the Hugs 1.3 implementation does not support the Haskell modules.

```
? tsigToStr (typeof (3::Int))
"Unknown.Int"
```

where `tsigToStr` computes the string representation of `TSignature` $a$. The string notation is not important as far as it uniquely represent a type.

The relation between `PerRoot` and `Typeable` is declared in the `PerRoot` class declaration like this:

```
> class Typeable a => PerRoot a where
```

In other words, `PerRoot` provides the root specification functionality, while `Typeable` abstracts the root identification procedure. Notice that the `Typeable` class automatically puts the same restriction posed by the `Ground` class (Section 4.2). Therefore, the `Ground` class is replaced with `Typeable` in the prototype implementation.

### 8.2.2 Automatic class-type instantiation

If the above instance declaration were required for every type constructor that may be a component of a database schema, it would be too burdensome for a database schema designer to do so. To lessen the burden, the script compiler has been modified so that every type constructor is automatically made an instance of an appropriate `TypeableCon`$i$ class.

This modification affects neigher the type checker nor the byte-code compiler. As noted in Chapter 2, a `deriving`-clause of a data type declaration specifies that the corresponding system-defined instance declaration is automatically derived by the language processor. Derived instances are generated by the phase, *static analysis*, before the type checking phase. The static analysis phase is aimed at converting a parsed program to the one suitable for the later phases. The automatic generation of the required `Typeable` class instances are performed in this phase.

The automatic instantiation of the `Typeable` class has another advantage. A user is prohibited to define his/her own instances for user-defined types, since Haskell precludes overrapping instance declarations. Declaring an instance $T$ of a class $C$ is permitted iff there is no other instance, say $T'$ of $C$ such that $T$ and $T'$ are unifiable. The current prototype automatically generates instantiation

$$(\texttt{Typeable}\ a_1, \cdots, \texttt{Typeable}\ a_n)\ \texttt{=>}\ \texttt{Typeable}\ (T\ a_1 \cdots a_n),$$

for every type constructor $T$, which follows the required property.

We shall end this subsection with one implementation-specific restriction. Some type constructors like `TypeableCon`$i$ take higher-order arguments. To illustrate more practical cases, consider two different tree types defined as follows:

```
> type BTree a = BLeaf a
>              | BBranch (BTree a) (BTree a)
>
> type GTree m a = GLeaf a
>                | GBranch (m (GTree m a))
```

The first example is a well-known binary tree structure with values only at the leaf level. The second example generalizes the concept so that the structure of branches are parameterized by the constructor variable `m`. Hence, `GTree []` is another type constructor for trees having lists as their branch structures, and so is `GTree BTree` except that the branches are binary trees. The strategy for automatic instance derivation described above does not work for these types, so the current prototype excludes such constructors in its automatic derivation phase of the instances of the `Typeable` class.

### 8.2.3 Structure of the root-value map

The database state includes an association which maps a persistent root designation to the corresponding value. The root value is represented by an *anonymous* cell. By anonymous, we mean that the value is coerced to an object of the `Any` type which have two internal operations:

```
> makeAny  :: a -> Any
> readAny :: Any -> a
```

which are identity functions internally. Note that `readAny` need *no* type-checking at run-time, because

- the garbage collector explores the heap cell structures directly, instead of checking auxiliary type information generated at compile time, and
- the root type annotation safely determines the type of the associated cell.

The type of the database state type is defined in the current prototype as follows:

```
> type RootMap    = [(String, Version, Any)]
> type RootMapObj = VObj RootMap
```

A value of `VObj` $a$ is a versioned object whose internal structure is a map from versions to values of type $a$. The details will be described in the next subsection. Thus the root-value map is an ordinary versioned object, except that the object is located by a designated internal variable.

The value is a list of tuples, and every tuple in the list corresponds to a persistent root[9]. The tuple associates the type annotation (the first tuple-element) to the anonymous value (the third tuple-element). The second element of the tuple is used only if the root is a view. It records the version in which the derived value is computed and is cached in the current map. Only if the required version is equal to this element, the cached value is used to save the recomputation cost.

[9]There is no explicit reason but simplicity to use the list type. A more efficiently accessible structure like a balanced tree could replace it.



## 8.3 Implementing versions

An object may have more than one values. Remember the object history shown in Fig. 5.1 (b) and Fig. 5.2. The object is created at $t_1$. The values of the object are assigned $v_3$ and $v_5$, respectively at $t_3$ and $t_5$. The values at other points are inherited according to the version generation history. For instance, the object value at $t_6$ is $v_5$, and the value at $t_7$ is $v_1$. Therefore, the following points are considered in implementation versions: how to represent versions, how to record history of modification, and how to cooperate the garbage collector with the bookkeeping mechanism.

### 8.3.1 Versioning and version representation

The object dereferencing procedure is implemented by a partial function from versions and object surrogates to values. This two-dimensional space may be represented either by projecting the space to one of the axes. Projecting to the version history axis (or time axis) associates the version with a collection of surrogate-value pairs, and the other projection associates every object with its own modification history. The advantages of this method is summarized as follows:

- Searching a value of a particular object for a certain version is a local search in the object modification history.
- A version can be represented by a time-stamp with a small bookkeeping record.
- In-place object update is easier to implement.

Every object carries its history of modification. For example, the history of the object shown in Fig. 5.2 can be depicted in a tabular form as follows:

| version | value |
|---|---|
| $t_1$ | $v_1$ |
| $t_3$ | $v_3$ |
| $t_5$ | $v_5$ |

The values at $t_5$, $t_3$ and $t_1$ are recorded in the history, while others are not. This implies that dereferencing a value for a particular version may require the reverse traversal of the version generation history. To support this traversal, the information record of a version has a reverse pointer to its parent. As shown in Fig. 5.1, the ancestors of the version $t_4$ is like this:

| version |
|---|
| $t_0$ |
| $t_1$ |
| $t_4$ |

.

"Joing" these two tables generates the following table

| version | value |
|---|---|
| $t_1$ | $v_1$ |

,

and the value at $t_4$ is the most recently generated value in this table, i.e., $v_1$.

The following is the code in Haskell to select a value of a particular object in a certain version:

```
> dereferObject version object
>   = do versionTable <- versionHistory version
>        objectTable  <- objectHistory  object
>        match <- return [ val |
>                          vsn         <- versionTable,
>                          (vsn', val) <- objectTable,
>                          vsn == vsn' ]
>        case match of
>           (val:_) -> return val
>           []      -> ... error dangling pointer ...
```

where `versionHistory` and `objectHistory` retrieve the *reverse* history information of the given version and the object, respectively. The computational complexity of the algorithm shown above is $O(m \times n)$, where $m$ is the height of the version tree, and $n$ designates the length of the object history. If the dereferencing procedure is implemented by an built-in operator, this search algorithm can be made faster: $O(m+n)$ by using two



traversal pointers. If the object history were represented by a hash table, then the order would be $O(m)$. The current prototype adopts the simple pointer traversal for brevity.

In addition to the parent pointer in the information are of a version, the number of locks held on the version is also recorded. The number is zero when the version is created, is incremented whenever `getDB`, `transaction`, or the rule-condition checker require "freezing" of the version. It is decremented when one of these frees it.

Updating an object is straightforward in this setting. A new entry for a version in an object history is created if needed, and otherwise the corresponding entry is overwritten. The consistency of the writer and readers is maintained in the implementation of the database monad, and is not taken care of by the internal storage manager.

### 8.3.2 Structure of the database state

In the Haskell part of the database monad implementation, a database state value comprises the original version, abort-request flag, and a job queue. The first component is required to abort the current transaction. The abort request flag is turned on by the `markAbortDB` function. The third component records the list of jobs that are executed at the transaction boundary. Note that the current version is not included in the state value. Instead, the version is recorded in a designated internal variable, and whenever the current transaction aborts, the original version replaces it.

### 8.3.3 Garbage collection

Checking liveliness of an object value is a little bit complicated in our case. Since the bookkeeping data have more than one pointer to a version, it is not always possible for the default garbage collector to discriminate garbage from live cells. The current prototype, hence, makes use of a conservative modification of the default garbage collector. We must take care of two kinds of liveliness as follows:

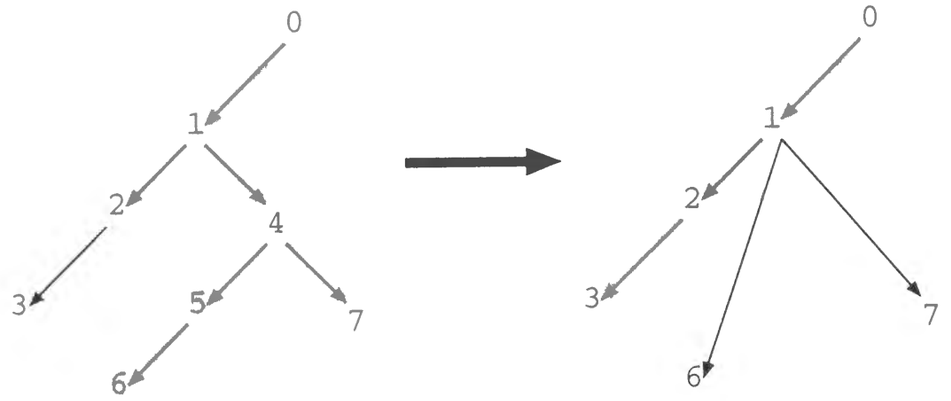



Figure 8.4: Reconstructing a version tree

- Heap cells but versions are live iff they are reachable from designated roots of the system; and
- Versions are live iff they are required from any of live objects.

To implement the second condition, the current conservative algorithm treats the parent pointer in versions as *weak*, where a weak pointer is treated as a normal pointer except that, during the liveliness-determination phase, they are not traversed. The version tree is reorganized in the garbage collection phase to eliminate unnecessary version nodes. To illustrate, consider the version tree shown in Fig. 5.1 again. If the versions 4 and 5 become garbage, the version tree is reorganized so that he version 1 becomes the parent of the versions 6 and 7. This process is illustrated in Fig. 8.4. Remember that the order of dereferencing is $O(m+n)$ where $m$ is the depth of the tree. Since $m$ affects the overall retrieving performance, this reorganization has the global effect on the database access performance.

Cutting unnecessary version nodes is also preferable from the view point of locality. An object history can be stored in an array, i.e., a contiguous chunk of cells in the flat space, so the constant factor of searching a particular version entry in the object history



is relatively small because of this locality. On the other hand, achieving the same locality for the version tree is impossible in general, even though a clustering algorithm might improve the locality.

The current garbage collection algorithm is conservative, since it does not reorganize object histories. An stricter algorithm might consider pointers from object histories to versions to be weak, and reorganize the object histories, too. This improved strategy, however, would complicate the garbage collector; since every object in a database might have to be reorganized. Besides, when reorganization of the version tree might duplicate a value entry in a object value history. In the example shown in Fig. 8.4, if an object has an entry for the version 4, the entry for the version 4 would have to be duplicated for the versions 6 and 7. This algorithm would cost more in time and space. Note that there is a space leak problem in the current conservative algorithm; even if a version is not required at all, it may never become garbage if one of the objects may have an entry for the version.

### 8.3.4 Comparison with the version-axis projection

As described above, the version-surrogate space is implemented through projecting it onto the surrogate-axis. Although it is impractical to decide which is the better without taking into account target applications, we shall compare the surrogate-axis projection with the version-axis projection to clarify the relative advantages of the current implementation.

Projecting onto the time-axis requires us to maintain trees as in the lazy state-transformer approach described in Section 5.6.1. This technique is easier to implement, since the bookkeeping routine does not have to keep track of the version history. Besides, the space leak problem pointed out above does not occur at least in the same form. This method, however, has a few drawbacks:

- The order of dereference is $O(\log s)$, where $s$ is the number of the stored surrogates.

This implies that the database access slows down according to the growth of the database population.

- The garbage collector must check the liveliness of objects, and reorganize versions to squeeze out dead objects.
- Ensuring in-place updatability is more difficult to achieve.

The third drawback could be avoided by using a reference counting garbage collector, which would require large cost of maintaining the reference counters though.

# Chapter 9

# An Example: the Train Database

Paton et al. (1996) used a database of British Railways for comparing the modeling capability and operations of miscellaneous persistent programming languages. This chapter also describes the implementation of the database to exhibit the proposed persistent programming environment in a more practical setting.

The train database comprises the station network and the train schedule mainly on Scotland area, UK. Fig. 9.1 exhibits the overall schema of the database in an enhanced entity-relationship diagram (Elmasri and Navathe 1994). The scheme diagram is straightforward, but we include brief explanation to clarify the meaning of the diagram:

- The station network is composed of various train routes.
- Stations are main or local. A main station may connect more than one route, while local stations may not.
- Trains are scheduled independently of the days of the week.

The following are the comments on the notations for unfamiliar readers:

- `MainStation` and `LocalStation` are subtypes of `Station`;
- The extent of `Station` is the disjoint union of those of `MainStation` and `LocalStation`.

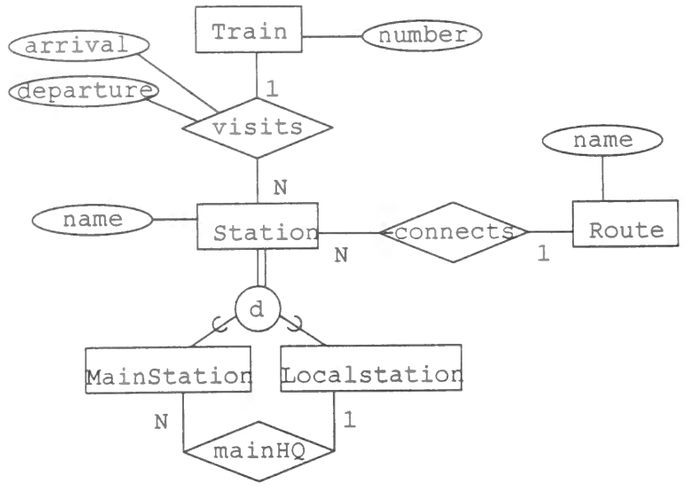



Figure 9.1: Schema of the train database.

The symbol, circled "d", represents the disjointness (there is no entity which belongs to the `MainStation` and `LocalStation` entity sets), and the double line between `Station` entity type and this circled symbol represents that every `Station` entity should be either a `MainStation` entity or `LocalStation` entity (*total specialization* constraint).

Note that the diagram does not necessarily model the real world precisely. The `visit` relationships for a particular train participant should be ordered appropriately on their arrival and departure time[1]. In this ordered set, the departure time of a station should follow the arrival time of the station (if any) and the arrival time should follow the departure time of the preseeding station (if any).

Another point which is not clearly captured in the diagram is how the entities are managed. The entity-relationship model assumes the type extent model of persistence, while we use the reachability model of persistence. Because of this difference, we have to decide the persistent root structures in designing the concrete database schema.

[1]The ordering might be captured better by an object-oriented modeling like OMT (Rumbaugh et al. 1991). As exhibiting structure is the main aim of this diagram, we follow a more conventional style of showing the data structures.



Usually, an abstract database schema described in a semantically rich database model is translated to a more concrete database schema in a implementation database model (Chen 1976; Hull and King 1987; Rumbaugh et al. 1991; Elmasri and Navathe 1994; Teorey 1994). Our implementation has not been facilitated with such methodologys for now. So the following explanation should be considered as an *ad hoc* translation of the abstract schema to the implementation schema.

The following sections show the entity design, the relationship design and the constraint enforcement, and some example queries for this database. Especially, the route search program, which searches between possible routes between two stations, will be shown. Note that we will not take into account the route entity type for brevity. The design strategy would not differ so much, though.

## 9.1 Designing Data types and persistent roots

The abstract schema includes three entity types. One of them, `Station` has two subtypes, `MainStation` and `LocalStation`. As has been done on the part-supplier database, this disjoint and total specialization relationship can be modeled naturally by an algebraic data type. Other possibilities would include (1) defining two data types for main and local stations, respectively, having a field which represents the shared information between two types of the stations, and (2) defining a single data constructor for stations which includes a field representing an additional information regarding the main or local station. The first alternative would be useful when the subtyping is *overlapping* rather than *disjoint*. The second choice would be made if the specialization were *partial* instead of *total*.

The relationship management is a little bit complicated task, mostly because bi-directional references are not supported in the proposed programming environment, while the concept of bi-directional relationships is supported in Object Definition Language

```
> data Station
>  = MainStation { stName  :: String,
>                  stStops :: [(DBRef Train, Maybe Time, Maybe Time)] }
>  | LoclStation { stName  :: String,
>                  stStops :: [(DBRef Train, Maybe Time, Maybe Time)],
>                  stNearestMn :: DBRef Station }
>
> data Train = Train { trNo:: Int,
>                      trVisits :: [(DBRef Station, Maybe Time, Maybe Time)]}
```

Figure 9.2: Data types for the train database.

defined by *Object Database Management Group* (ODMG) (Cattel and Barry 1997)[2]. Instead of extending the modeling capability of the programming environment, the proposed methodology makes use of the the triggering mechanism to manage the relationships. We will discuss the topic later, and show the data type declaration part here (Fig. 9.2).

The definitions utilize `Time`, another user-defined algebraic data type, and `Maybe`, one of the system-supplied data type. The details of `Time` are not important though. For brevity, we only note that the following functions can be used to handle time data

```
> mkTime :: String -> Time
```

and that the type is an instance of *Eq*, *Ord*, *Num*, and *Show*. This instantiation make the following operations available for *Time*:

- *Eq*: `==`, `/=`,
- *Ord*: `>`, `>=`, `<`, `<=`,
- *Num*: `+`, `-`,
- *Show*: `show`,

where `show` maps a value to its string representation.

[2]In the ODMG Standard, the schema designer can specify that an attribute value of an object is the inverse of an attribute value of another object.



```
> data StationMap = StationMap [(String, DBRef Station)]
> instance PerRoot StationMap where
>    initValue _ = StationMap []
>
> data TrainMap = TrainMap [(Int, DBRef Train)]
> instance PerRoot TrainMap where
>    initValue _ = TrainMap []
```

Figure 9.3: persistent root types for the train database.

The `Maybe` type is defined as follows:

```
> data Maybe a = Just a | Nothing
```

This type is used to represent arrival and departure times of trains. The following is a visiting schedule of a train from Aberdeen to London:

| | | |
|---|---|---|
| Aberdeen | –:– | 10:00 |
| Edinburgh | 13:00 | 13:00 |
| York | 15:00 | 15:00 |
| London | 17:00 | –:– |

where "–:–" represents that there is no corresponding time. `Just` and `Nothing` are used to represent, respectively, explicit time entries and "–:–" entries.

The next step of the design is to define persistent roots. In the running example (the part-supplier database) persistent roots were the lists of surrogates. For the train database applications, however, it is more convenient to treat key-to-surrogate maps as the persistent roots, because they are often accessed by their key values. Fig. 9.3 shows the definitions of the persistent roots.

The key-to-surrogate maps might be defined using the view mechanism of the proposed framework. The access performance would not degrade so much because of the cache mechanism. For comparison, Fig. 9.4 exhibits the view style definitions of the key-to-surrogate maps, provided that the persistent roots for the lists of identifiers be maintained.

```
> {-
>  - we assume that the identifiers are maintained by
>  - the roots associated with [DBRef Station] and [DBRef Train],
>  - respectively.
>  -}
>
> data StationMap = StationMap [(String, DBRef Station)]
> instance PerRoot StationMap where
>    isView _ = True
>    initValue db
>       = StationMap [ (stName (readRef db sid), sid) | sid <- getRoot db ]
>
> data TrainMap = TrainMap [(Int, DBRef Train)]
> instance PerRoot TrainMap where
>    isView _ = True
>    initValue db
>       = TrainMap [ (trNo (readRef db tid), tid) | tid <- getRoot db ]
```

Figure 9.4: persistent root types for the train database with the views.

## 9.2 Designing entities and triggers

Now that the extent structure has been defined, the entity definitions follow. The minimum requirements of the train database are to maintain the persistent roots automatically, and to enforce the mutual reference integrity constraints between stations and the trains.

Fig. 9.5 defines the `Entity-Station` instantiation. The roles of the declared instance methods are summarized as follows:

- `beforeNew` and `beforeUpdate` check if the station to be inserted or modified has a unique name in the current known stations;
- `afterNew` inserts the newly created station entity into the persistent root; and
- `beforeUpdate` prohibits a main station from being graded down to a local station[3].

[3]This would be important if another constrain that any route should start and end with main stations were added.



```
> instance Entity Station where
>     -- check key constraint.
>     beforeNew station
>       = do StationMap stByName <- readRootDB
>            case [ sid' | (stName', sid') <- stByName, stName' == sname] of
>              [] -> return ()
>              _  -> error ("beforeNew[Station]: Station name \""
>                                 ++ sname ++ "\" duplicates.")
>       where sname = stName station
>     -- maintain the type extent
>     afterNew sid station
>       = do StationMap stByName<- readRootDB
>            writeRootDB (StationMap ((stName station, sid):stByName))
>
>     -- modification check.
>     beforeUpdate sid oldVal newVal
>       = do -- (1) prohibit main to local type change.
>            prohibitStationTypeChange oldVal newVal
>            -- (2) maintain the name map.
>            StationMap stByName <- readRootDB
>            (case [ sid' | (stName', sid') <- stByName,
>                           stName' == stNameNew, sid' /= sid ] of
>               [] -> return ()
>               _  -> error ("beforeUpdate[Station]: Station name \""
>                                ++ stNameNew ++ "\" duplicates."))
>       where
>         stNameOld = stName oldVal
>         stNameNew = stName newVal
>         prohibitStationTypeChange MainStation{} LoclStation{}
>           = error "Changing a main station to a local station is prohibited."
>         prohibitStationTypeChange _ _ = return ()
```

Figure 9.5: Definition of the station entity type

```
> instance Entity Train where
>    -- check key constraints
>    beforeNew Train{trNo}
>      = do TrainMap trByNo <- readRootDB
>           ( case [ tid' | (trNo', tid') <- trByNo, trNo == trNo'] of
>              [] -> return ()
>              _  -> error ("beforewNew[Train]: train no \""
>                           ++ show trNo ++ "\" duplicates."))
>
>    -- maintain the type extent and the inverse function.
>    afterNew tid Train{trNo,trVisits}
>      = do -- (1) maintain the type extent
>           TrainMap trByNo <- readRootDB
>           writeRootDB (TrainMap ((trNo, tid):trByNo))
>           -- (2) maintain the inverse function.
>           addInverseVisits tid trVisits
>
>    -- check key constraints
>    beforeUpdate tid Train{trNo=trNoOld} Train{trNo=trNoNew}
>      = do TrainMap trByNo <- readRootDB
>           ( case [ tid' | (trNo', tid') <- trByNo,
>                           trNo' == trNoNew, tid' /= tid ] of
>              [] -> return ()
>              _  -> error ("beforewUpdate[Train]: train no \""
>                           ++ show trNo ++ "\" duplicates."))
>
>    afterUpdate tid Train{trVisits=trVisitsOld}
>                    Train{trVisits=trVisitsNew}
>      = do delInverseVisits tid trVisitsOld
>           addInverseVisits tid trVisitsNew
```

Figure 9.6: Definition of the train entity type



```
> delInverseVisits tid trVisits
>   = sequence
>       [ do station <- readDB sid
>            let tt = stStops station
>                newStops = [ sched | sched@(tid', _, _) <- tt, tid' /= tid ]
>            writeDB sid station{stStops=newStops}
>         | (sid, _, _) <- trVisits ]
>
> addInverseVisits tid trVisits
>   = sequence
>       [ do station <- readDB sid
>            let tt = stStops station
>                newStops = sortByTime ((tid, atime, dtime) : tt)
>            writeDB sid station{stStops=newStops}
>         | (sid, atime, dtime) <- trVisits ]
>
>   where
>     sortByTime [] = []
>     sortByTime ((tid,atime,dtime):xs)
>       = list1 ++ [(tid,atime,dtime)] ++ list2
>       where list1 = [ visit | visit@(tid',atime',dtime') <- xs,
>                               sortTime atime' dtime' <= sortTime atime dtime ]
>             list2 = [ visit | visit@(tid',atime',dtime') <- xs,
>                               sortTime atime' dtime' >  sortTime atime dtime ]
>     sortTime Nothing      Nothing      = error "illegal time spec."
>     sortTime Nothing      (Just time)  = time -- departure time only
>     sortTime (Just time)  Nothing      = time -- arrival time only
>     sortTime (Just atime) (Just dtime) = atime -- both are considered.
```

Figure 9.7: Utility functions for the relationship management

Note that there is no consistency checking for the trains that stop at a station. This property is handled in maintaining trains. The `Train` type is defined as shown in Fig. 9.6. The difference from the `Station` type is that it makes use of two utility functions, `delInverseStops` and `addInverseStops`, to maintain relationships between `Station` entities and `Train` entities. Whenever a train schedule changes, the corresponding stopping schedules of stations are changed accordingly. Although the definitions of the utility functions are straightforward, we include them in Fig. 9.7 for completeness and showing the functional flavor of the programming. Especially, `sortByTime` is a well-known code for the quick sort algorithm (Bird and Wadler 1988; Davie 1992).

## 9.3 Populating the database

The next step is to populate the database. This process is split up into two steps reflecting our decision that relationships between stations and trains are maintained on the station side. So the first step inserts all the stations, and the train extent is populated next. Figs. 9.8, 9.9, and 9.10 show the initialization code. For reader's help, the network of the stations included in the database is shown in Fig. 9.11 The full initialization code is listed in Appendix B.1. The function `initdb` performs a single transaction, which inserts stations and trains. We only have to define entities here; the automatic execution of `beforeNew` and `afterNew` methods maintains the persistent roots and the relationships between trains and stations appropriately.

Another point that should be noted here is that we use `TimeType` to discriminate the stop types. This reflects the fact that a train's relationship with a station can be categorized into four groups respectively represented by the following data constructors:

- `Dprt`: the station is the train's departing point;
- `Arvl`: the station is the train's final destination;



```
> type STime = String
> data TimeType = Arvl STime | Dprt STime | Stop STime | ArDp STime STime
> initdb = transaction initStationsAndTrains
>
> initStationsAndTrains =
>   do -- main stations
>      aberdeen     <- mkMain "Aberdeen"
>      inverness    <- mkMain "Inverness"
>      edinburgh    <- mkMain "Edinburgh"
>      glasgow      <- mkMain "Glasgow"
>      london       <- mkMain "London"
>      southampton  <- mkMain "Southampton"
>      -- local stations
>      nairn        <- mkLocl "Nairn" inverness
>      aviemore     <- mkLocl "Aviemore" inverness
>      kingussie    <- mkLocl "Kingussie" inverness
>      pitlochry    <- mkLocl "Pitlochry" inverness
>      perth        <- mkLocl "Perth" edinburgh
>      stirling     <- mkLocl "Stirling" edinburgh
>      falkirk      <- mkLocl "Falkirk" edinburgh
>      aberdour     <- mkLocl "Aberdour" edinburgh
>      banchory     <- mkLocl "Banchory" aberdeen
>      prestonpans  <- mkLocl "Prestonpans" edinburgh
>      northBerwick <- mkLocl "North Berwick" edinburgh
>      york         <- mkLocl "York" edinburgh
>      winchester   <- mkLocl "Winchester" southampton
```

Figure 9.8: Initialization code: populating stations

```
>     ------------------
>     -- add train
>     ------------------
>     -- (1) Trains from and around Aberdeen.
>     --  aberdeen to edinburgh
>     mkTrain 22403101 [ (aberdeen   ,Dprt "07:20"),
>                        (edinburgh  ,Arvl "09:50") ]
>     mkTrain 22407101 [ (aberdeen   ,Dprt "10:00"),
>                        (edinburgh  ,Stop "13:00"),
>                        (york       ,Stop "15:00"),
>                        (london     ,Arvl "17:00") ]
>     mkTrain 22403102 [ (aberdeen   ,Dprt "14:00"),
>                        (edinburgh  ,Arvl "16:30") ]
>
>     -- aberdeen to inverness
>     mkTrain 22446301 [ (aberdeen   ,Dprt "08:00"),
>                        (inverness  ,Arvl "10:45") ]
>     mkTrain 22446302 [ (aberdeen   ,Dprt "12:00"),
>                        (inverness  ,Arvl "14:45") ]
>     mkTrain 22446303 [ (aberdeen   ,Dprt "19:00"),
>                        (inverness  ,Arvl "21:45") ]
>
>     -- aberdeen to banchory
>     mkTrain 22490101 [ (aberdeen   ,Dprt "08:30"),
>                        (banchory   ,Arvl "08:50") ]
>     mkTrain 22490102 [ (aberdeen   ,Dprt "12:00"),
>                        (banchory   ,Arvl "12:20") ]
>     mkTrain 22490103 [ (aberdeen   ,Dprt "18:00"),
>                        (banchory   ,Arvl "18:20") ]
>
>     .....
```

Figure 9.9: Initialization code (continued): populating stations



```
>   where
>     mkMain name
>       = newDB MainStation{stName=name, stStops=[]}
>     mkLocl name mn
>       = newDB LoclStation{stName=name, stStops=[], stNearestMn=mn}
>     mkTrain no visits
>       | ( case snd (head visits) of
>             Dprt _ -> True
>             _      -> False ) &&
>         ( case snd (last visits) of
>             Arvl _ -> True
>             _      -> False ) &&
>         ( if length visits >= 2 then
>             foldr (&&) True
>               [ case sched of {
>                   Stop _ -> True; ArDp _ _ -> True; _ -> False }
>                 | (_, sched) <- take (length visits - 2) (tail visits) ]
>           else True )
>       = newDB Train{trNo=no, trVisits=visits'}
>       where visits'
>               = [ (st, mkArr sched, mkDep sched ) | (st,sched) <- visits ]
>             mkArr (Dprt time)   = Nothing
>             mkArr (Stop time)   = Just (mkTime time)
>             mkArr (ArDp time _) = Just (mkTime time)
>             mkArr (Arvl time)   = Just (mkTime time)
>             mkDep (Dprt time)   = Just (mkTime time)
>             mkDep (Stop time)   = Just (mkTime time)
>             mkDep (ArDp _ time) = Just (mkTime time)
>             mkDep (Arvl time)   = Nothing
```

Figure 9.10: Initialization code (continued): utility functions

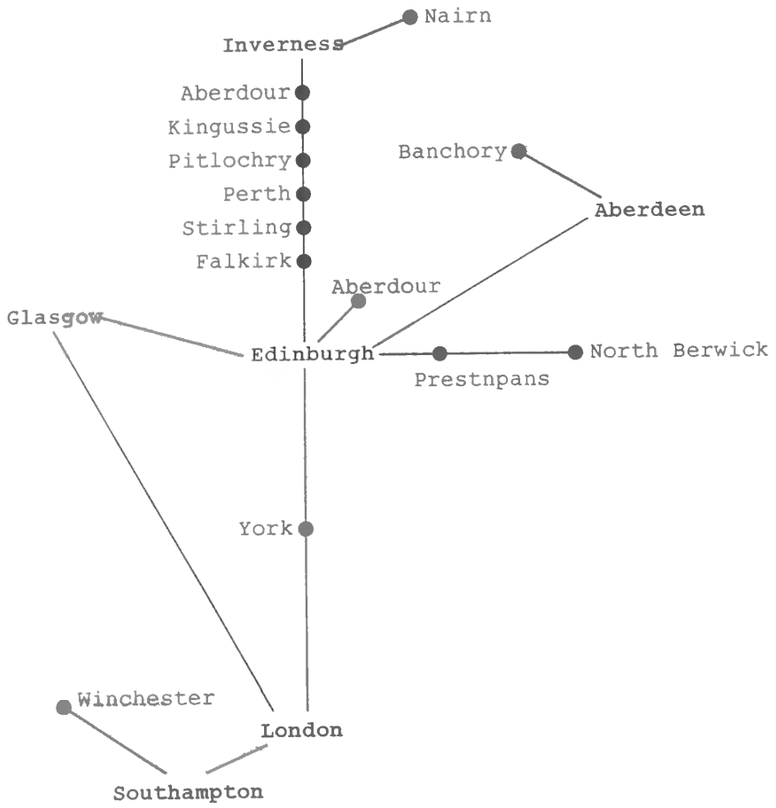



Figure 9.11: Topological relationships of the stations; this figure is a redrawing of Figure 2.10 in Paton et al. (1996).



- `Stop`: the train makes a brief stop at the station; and
- `ArDp`: the train makes a longer stop at the station worthwhile recording in the time chart.

The utility functions improve the readability of the input data. `mkMain` makes a `MainStation` data structure, and inserts it into the database. `mkLocl` is a similar one. `mkTrain` checks if the given stopping schedule is valid; the stopping schedule should start with a departing point and end with a final destination through stations where the train makes a stop.

## 9.4 Searching routes between two stations

The typical query searches routes between two given stations under certain constraints. The following may be some of plausible conditions:

- The time for transit must be within a particular duration.
- Visiting a station twice should be avoided.
- The total travel time should be restricted.
- The result should be answered before a particular elapsed time.
- The total travel fee should be under a particular amount.

Other conditions might be considered. We adopts the first two conditions only here, even though the others could be taken into account without difficulty.

We exhibit one of the simplest algorithms, which traverses the train network from the given source station until the final destination is found. More intelligent method would be (1) considering the geographical locations of the stations, or (2) finding the nearest main stations of the source and the destination stations and considering the only combination of direct connection between the main stations. The simplest form of the

```
? findroute "Inverness" "Banchory"
... searching routes from Inverness to Banchory
---------------------------------------
Inverness      : --:--, 09:00: 46322401
Aberdeen       : 11:45, --:--: 46322401
Aberdeen       : --:--, 12:00: 22490102
Banchory       : 12:20, --:--: 22490102
---------------------------------------
Inverness      : --:--, 10:15: 46303102
Edinburgh      : 14:15, --:--: 46303102
Edinburgh      : --:--, 15:00: 07122402
Aberdeen       : 17:45, --:--: 07122402
Aberdeen       : --:--, 18:00: 22490103
Banchory       : 18:20, --:--: 22490103
```

Figure 9.12: All the routes from Inverness to Banchory

algorithm, however, exhibits the computational strength of the functional programming language using recursive programming, and the author believes that readers who are not familiar with Haskell can understand the query function without difficulty.

Fig. 9.12 shows the routes between Inverness to Banchory. The first route visits Aberdeen for connection, and the second one is via Edinburgh. The full listing of the query function is shown in Appendix B.2. The design of the query starts from the top-

```
> dofindroute db [] to
>   = []
> dofindroute db routes to
>   = reached ++ findMore
>   where reached    = [ path | path@((sid, _, _, _):_) <- routes, sid == to ]
>         notReached = [ path | path@((sid, _, _, _):_) <- routes, sid /= to ]
>         nextRoutes = do path <- notReached
>                         findNext db path to
>         findMore = dofindroute db nextRoutes to
```

Figure 9.13: Function to find routes



```
> findNext ::
>    Database -> [(DBRef Station, DBRef Train, Maybe Time, Maybe Time)]
>             -> DBRef Station
>             -> [[(DBRef Station, DBRef Train, Maybe Time, Maybe Time)]]
> findNext db [] to
>   = []
>
> findNext db path@((sid, tid, arrive, depart):_) to
>   -- has arrived at the destination
>   | sid == to
>   = [path]
>
>   -- departs from this station.  Add the next stop.
>   | arrive == Nothing || length path == 1
>   = [ (sid', tid, arr', dep'): path
>       | (sid', arr', dep') <- nextStop sid tid,
>         not (sid' `elem` map (\(x,_,_,_) -> x) path) ]
>
>   -- arrives here but requires further travel.
>   | otherwise
>   =  -- (1) with the same train.
>     [ (sid', tid, arr', dep'): path
>       | (sid', arr', dep') <- nextStop sid tid,
>         not (sid' `elem` map (\(x,_,_,_) -> x) path) ]
>  ++ -- (2) via another train.
>     [ (sid2, tid1, arr2, dep2):(sid, tid1, arr1, dep1): path
>       | (tid1, arr1, dep1@(Just dtime1)) <- stStops (readRef(db) sid),
>         tid1 /= tid,
>         dtime1 - atime <= mkTime "01:00",
>         dtime1 - atime >= mkTime "00:10",
>         (sid2, arr2, dep2) <- nextStop sid tid1,
>         not (sid2 `elem` map (\(x,_,_,_) -> x) path) ]
>
> where
>   Just atime = arrive
>   Just dtime = depart
>   nextStop sid tid
>    = let Train{trNo,trVisits} = readRef(db) tid
>          sched = dropWhile (\(sid', _, _) -> sid' /= sid) trVisits
>      in if length sched == 0
>         then error ("dofindroute.nextStop: train " ++ show trNo ++
>            " does not stop at \"" ++ stName (readRef(db) sid) ++ "\".\n")
>         else take 1 (tail sched) --- maybe [], if no next stops.
```

Figure 9.14: Function to find the next stop

most function, `findroute`, which calls the search function. Other roles of this top-most function are to check the validity of the input station names, and to print the search result in a user-friendly way by calling `printRoute`. The retrieval procedure makes use of the on-the-fly dereferences. The current state retrieved by `getDB` is passed to the graph traversal function to perform lazy retrieval of the database state.

The main function, `dofindroute`, performs the edge-wise graph search according to the train schedule. Fig. 9.13 exhibits the code. Because of the semantics of the list comprehension (Wadler 1987), the algorithm itself specifies the depth-first-search of the train network. Some optimization technique, however, would automatically improve the search strategy (Trinder 1991), or make use of a general-purpose graph search algorithms (Launchbury 1995).

The search strategy is very simple. Given a known set of paths from the source station, the function splits the path into the "reached" and "non-reached" paths. For each of the latter paths, the function calls another function, `findNext`, shown in Fig. 9.14 to move the "traveler" along the train network. The `findNext` function reflects the search strategy that considers four cases where the traveler, respectively,

- has reached the destination;
- has just departed the source station;
- is on the way to the destination, and takes the same train; and
- is on the way to the destination, and takes another train which satisfies the connection condition.

In this example, the connection condition is that the time for the transit is between ten minutes to one hour[4]. Besides this condition, the cyclic route is prohibited for any path. The predicates in the list comprehensions such as

[4]The example can not handle "sleepers".



```
not (sid' `elem` map (\(x,_,_,_) -> x) path)
```

ensure the condition.

The first argument of these functions (`db`) is of type `Database` that is obtained in the top-most function (`findroute`) and is passed to them. As the abstract entities are represented by their surrogates, their values are retrieved using `readRef(db)`. This is a little bit onerous in writing functions, but the equality checking is faster and easier to manage than that based on the structural equivalence.

# Chapter 10

# Summary and Future Work

A persistent programming environment for the standard, non-strict, purely functional language, Haskell, has been proposed. This chapter summarizes the notable features of the approach, and then addresses the further research issues.

## 10.1 Summary

### Monad-based approach without compromising purity

The monad-based update mechanism allows users to update the database state without compromising the purity of the paradigm. Moreover, since the update unit here is an association of an object surrogate and its value, the name-equivalence is supported.

### Multiple versions to ease the burden of writing queries

Multiple database versions can be manipulated simultaneously, and also be locked to ensure lazy retrieve from the state. This consistently allows users to update database in-place, and also allows them to write query expressions making use of laziness of the language. Moreover, the "what-if" semantics of execution and non-materialized views are naturally supported.

### Persistent root identification based on type classes

The persistency identification is performed based on the types of the persistent roots, so every piece of the expressions including those manipulating persistent roots is statically typed. The persistent roots are defined in a declarative manner, making it unnecessary to "snooping" other programs to get information of the roots structures.

### Views are special types of persistent roots

Views are just special types of persistent roots. The different points are, (1) they are not allowed to be updated directly, (2) the initial value of a view root should be the expression that computes the view value. A computed view value may be cached in the current database state, which supports full laziness, i.e., just on-need and just once computation of view values.

### Triggering mechanism to maintain integrity constraints

The database entity types and persistent root types are defined through making them instances of the `Entity` or `PerRoot` classes, respectively. This allow programmers to add "hooks" for the primitive database operators to enforce integrity constrains such as mutual references, emulation of the type extent model of persistency, and handling dangling references.

### Support of transaction-boundary job-execution

As the language is higher-order, jobs that are executed at the transaction-boundary can be registered in the database state. With the help of multiple database versions, this rule-based computation supports the fixed point iteration as well as the usual queue-based iteration.



### Using a standard language

Besides the above advantages, it should be pointed out that Haskell is the standard language. The proposed approach can be integrated with a standard Haskell environment with a slight modification of the language processor, even though the storage mechanism should be modified so much. This means that the libraries for the language shared or sold by the third parties may be made use of without difficulty. Such libraries include, for example, graphics packages (Hallgren and Carlson 1995; Finne and Peyton Jones 1996), a graph traversal package (Launchbury 1995), and a parser generator (Gill and Marlow 1996)[1].

### Notes on the prototype

To show the feasibility of the environment, a prototype has been implemented by porting Hugs 1.3, a Haskell-compliant successor of Gofer (Jones 1994a), under Texas Persistent Store 0.5 (Singhal et al. 1992)[2]. The current prototype has implemented all the proposed features. The details of the modification has been addressed in Chapter 8, and the train database example has been described in Chapter 9. Although the garbage collection procedure is not efficient nor strict enough, the author believes that the minimum feasibility of the lazy functional programming for database manipulation has been shown.

## 10.2 Further research issues

Several important issues are yet to be explored in order to make the programming environment more practical.

[1] Regrettably, these packages depend too much on a certain implementation of the language processor, in spite of the standardization effort of the language.

[2] The first prototype described in (Ichikawa 1995) was developed using Glasgow Haskell Compiler 0.29 (Hall et al. 1993) with C procedures to manage database type extent.

### Relationship to database modeling methodologies

In the first phase of database management process, the real world is modeled using a semantically rich data model. The author has not yet facilitated the proposed environment with any conceptual design methodology. The database research community, however, has seen intensive research activities in this area. The semi-automatic translation of the entity-relationship model to the relational model has been studied intensively (Chen 1976; Teorey 1994; Elmasri and Navathe 1994). The recent trend in object-orientation has led to the development of Object Definition Language (ODL) (Cattel and Barry 1997), which models the real world in object-oriented terms independent of specific implementation programming languages. Ceri and Fraternali (1997) cover various topics in the conceptual design phase with the object-oriented or semantic database models and mapping abstract schemas to those in implementation models. FDL (Poulovassilis and Kind 1990) and its successor PFL (Small and Poulovassilis 1991; Small 1993) directly support the functional data model proposed by Shipman (1981).

### Active rule formalization

The proposed manipulation of active rules is not based on a mathematical foundation or formal model. Instead, incorporating the functionality has been made possible through the ability to handle multiple versions, the inherent computational completeness, and the class mechanism to support customizable overloaded functions. There are some known proposals of formal active database models based on delta-state (Ghandeharizadeh et al. 1996; Doherty et al. 1996), logic (Fraternali and Tanca 1995), and functions (Reddi et al. 1995; Poulovassilis et al. 1996). Reddi et al. (1995) and Poulovassilis et al. (1996) address the technique that allows execution of any user-defined function to be treated as an event. Even though this direction is practical for dedicated functional database programming languages, it is not necessarily clear that this is so in the persistent



programming environment extending a volatile programming language.

### Storage management and concurrency control

The storage manager of the prototype is not tuned for a persistent programming environment. In particular, the garbage collection algorithm for multiple versions has not been fully investigated yet. Another issue is the concurrency control. Even if the prototype were developed in a multi-user object-oriented programming environment such as Shore (Carey et al. 1994), the high update-rate due to garbage collection and lazy evaluation would make it difficult to allow for concurrent updating of the database storage. Remember that every suspended expression is updated whenever the value of the expression is reduced, and that partial application also may allocate heap cells.

### Optimization

One of the preferable features of the purely functional computation paradigm is its optimizability based on equational reasoning. As a result of this property, the functional programming language community has developed many optimization techniques for volatile functional languages. (See e.g., Peyton Jones and Lester (1992) for the pointers). On the other hand, the database community has made use of this property to devise optimization techniques that can typically be seen in Trinder (1991), Buneman et al. (1994) and Poulovassilis and Small (1996). We have to devise an optimization procedure that not only improves the performance of functional computation but also takes into account the physical data independence and the properties of storage devices. These two aspects are discussed in the database and functional programming language communities, and have not been explored well in conjunction with each other.

### Schema evolution and reflection

The current prototype does not handle schema evolution. The only method of modifying schema is to initialize the database state and the script defining the database schema, even though utility functions and query functions are freely redefinable. Staple (McNally and Davie 1991) handles schema evolution by retaining old modules as they are. Although this is a plausible approach, users must always be aware of which modules are accessed through persistent roots. Another approach uses dynamic typing and dynamic module binding as in Napier88 (Dearle et al. 1989).

Another aspect of the persistent programming is the reflection mechanism. As has been noted in the context of Napier88 by Kirby (1992), and has also been noted in a number of standard database texts, run-time accessibility to the meta level information, or schema information, is useful for users and for certain applications like graphical database query interfaces.

### Feasibility for advanced database applications

Practical targets of the persistent programming languages would be more complex than the running example, the part-supplier database. Multimedia applications would require more sophisticated user interfaces, and put more stress on computation resources. These could be considered as the issues for applying purely functional programming for realistic applications. Unless these issues are explored, however, the persistent programming environment based on the paradigm will not become a practical approach.

Besides the issues particularly found in monolithic environments, practical applications require cooperative tasks supported by miscellaneous tools. Such tools include spreadsheets, scientific or information visualization systems, high-performance computers, decision making systems, and, of course, heterogeneous and/or legacy information management systems. A persistent programming environment, Tycoon (Matthes and



Schmidt 1995) proposed a "gateway" approach, in which the persistent programming language plays the role of resolvent for existing tools. Although it is not sure whether a purely functional persistent environment can play the same role in a scalable information processing environment, the mathematical purity is expected to help information system designers specify the cooperation among them in a more formal way.

# Bibliography

Abadi, M., L. Cardelli, B. C. Pierce, and G. D. Plotkin (1991, April). Dynamic typing in a statically typed language. *Transactions on Programming Languages and Systems 13*(2), 237–268.

Achten, P., J. van Groningen, and R. Plasmeijer (1993). High level specification of I/O in functional languages. In J. Launchbury and P. Sansom (Eds.), *Functional Programming, Glasgow 1992*, pp. 1–17. Springer-Verlag.

Agrawal, R. and N. H. Gehani (1989, May). ODE (object database and environment): The language and the data model. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pp. 36–45.

Akerholt, G., K. Hammond, S. L. Peyton Jones, and P. W. Trinder (1993). Processing transactions on GRIP: A parallel graph reducer. In *Proceedings of PARLE '93*, pp. 634–647. Lecture Notes in Computer Science 694, Springer-Verlag.

Argo, G., J. Hughes, P. Trinder, J. Fairbairn, and J. Launchbury (1990). Implementing functional databases. In F. Bancilhon and P. Buneman (Eds.), *Advances in Database Programming Language*, pp. 165–176.

Atkinson, M. and R. Morrison (1995, July). Orthogonally persistent object systems. *The VLDB Journal 4*(3), 319–402.

Atkinson, M. P., P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison (1983). An approach to persistent programming. *Computer Journal 26*(4), 360–365.

Atkinson, M. P. and P. Buneman (1987, June). Types and persistence in database programming languages. *ACM Computing Surveys 19*(2), 105–190.

Bancilhon, F., C. Delobel, and P. Kanellakis (Eds.) (1992). *Building an Object-Oriented Database System: The Story of $O_2$*. Morgan Kaufmann.

Barbedette, G. (1992). Lisp $O_2$: A persistent object-oriented lisp. In *Bancilhon et al. (1992), Chapter 10*, pp. 215–233.

Bernstein, P. A., V. Hadzilacos, and N. Goodman (1987). *Concurrency Control and Recovery in Database Systems*. Addison Wesley.

Bird, R. and P. Wadler (1988). *Introduction to Functional Programming*. Prentice Hall.

Buneman, P., L. Libkin, D. Suciu, V. Breazu-Tannen, and L. Wong (1994, March). Comprehension syntax. *ACM SIGMOD RECORD 23*(1), 87–96.

Carey, M. J., D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling (1994, May). Shoring up persistent applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pp. 383–394.

Cattel, R. G. G. and D. K. Barry (Eds.) (1997). *Object Database Standard: ODMG 2.0* (2nd ed.). Morgan Kaufmann.

Ceri, S. and P. Fraternali (1997). *Designing Database Applications with Objects and Rules: the IDEA Methodology*. Addison-Wesley.

Chen, P. P.-S. (1976, March). The entity-relationship model — toward a unified view of data. *ACM Transactions of Database Systems 1*(1), 9–36.

Christophides, V., S. Abiteboul, S. Cluet, and M. Scholl (1994, May). From structured documents to novel query facilities. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pp. 313–324.



Christophides, V., S. Cluent, and G. Moerkotte (1996, May). Evaluating queries with generalized path expressions. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pp. 413–422.

Connor, R., D. McNally, and R. Morrison (1991). Subtyping and assignment in database programming language. In P. Kanellakis and J. W. Schmidt (Eds.), *Proceedings of the Third International Workshop on Database Programming Languages*, pp. 363–382. Morgan Kaufmann.

Consens, M. P. and T. Milo (1994, May). Optimizing queries on files. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pp. 301–312.

Date, C. J. and H. Darwen (1997). *A Guide to the SQL Standard* (4th ed.). Reading, MA: Addison-Wesley.

Davie, A. J. T. (1992). *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press.

Dearle, A., R. Connor, F. Brown, and R. Morrison (1989). Napier88 — a database programming language? In R. M. Richard Hull and D. Stemple (Eds.), *Proceedings of the Second International Workshop on Database Programming Languages*, pp. 179–195. Morgan Kaufmann.

Doherty, M., R. Hull, and M. Rupawalla (1996, June). Structures for manipulating proposed updates in object-oriented databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pp. 306–317.

Elmasri, R. A. and S. B. Navathe (1994). *Fundamentals of Database Systems* (2nd ed.). CA: Benjamin/Cummings.

Eswaran, K. P. and D. D. Chamberlain (1975, September). Functional specifications of a subsystem for data base integrity. In *Proceedings of the First International*

*Conference on Very Large Data Bases*, pp. 48–68.

Finne, S. and S. L. Peyton Jones (1996). Composing user interfaces with haggis. In J. Launchbury, E. Meijer, and T. Sheard (Eds.), *Adavanced Functional Programming: Second International Spring School on Advanced Functional Programming Techniques, Tutorial Text*. Lecture Notes in Computer Science 1129, Springer-Verlag.

Fishman, D. H., D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan (1987, January). Iris: An object oriented database management system. *ACM Transactions on Office Information Systems 5*(1), 48–69.

Flickner, M., H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker (1995, September). Query by image and video content: The QBIC system. *IEEE Computer 28*(9), 23–32.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence 19*(1), 17–37.

Fraternali, P. and L. Tanca (1995, December). A structured approach for the definition of the semantics of active databases. *ACM Transactions on Database Systems 20*(4), 414–471.

Gamerman, S., C. Lanquette, and F. Vélez (1992). Using database applications to compare programming languages. In *Bancilhon et al. (1992), Chapter 13*, pp. 278–324.

Ghandeharizadeh, S., R. Hull, and D. Jacobs (1996, September). Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Transactions on Database Systems 21*(3), 370–426.

Gill, A. and S. Marlow (1996, February). Happy manual, version 0.9. Technical report,



Department of Computer Science, University of Glasgow.

Gordon, A. D. (1994). *Functional Programming and Input/Output*. Cambridge University Press, Distinguished Dissertations in Computer Science.

Hall, C., K. Hammond, W. Partain, S. L. P. Jones, and P. Wadler (1993). Glasgow Haskell Compiler: A retrospective. In J. Launchbury and P. Sansom (Eds.), *Functional Programming, Glasgow 1992*, pp. 62–71. Springer-Verlag.

Hallgren, T. and M. Carlson (1995). Programming with Fudgets. In J. Jeuring and E. Meijer (Eds.), *Adavanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques, Tutorial Text*, pp. 137–182. Lecture Notes in Computer Science 925, Springer-Verlag.

Hammond, K., D. McNally, P. M. Sansom, and P. Trinder (1993). Improving persistent data manipulation for functional language. In J. Launchbury and P. Sansom (Eds.), *Functional Programming, Glasgow 1992*, pp. 72–84. Springer-Verlag.

Holyer, I. (1991). *Functional Programming with Miranda*. Pitman.

Hudak, P., S. L. P. Jones, and P. Wadler, eds. (1992, May). Report on the functional programming language Haskell, version 1.2. *ACM SIGPLAN Notices 27*(5).

Hughes, J. (1989). Why functional programming matters. *The Computer Journal 32*(2), 98–107.

Hull, R. and R. King (1987, September). Semantic data modeling: Survey, applications and research issues. *ACM Computing Surveys 19*(3), 201–260.

Ichikawa, Y. (1995, September). Database states in lazy functional programming languages: Imperative update and lazy retrieval. In *Proceedings of the Fifth International Workshop on Database Programming Languages*, pp. 150–163. (The final revision is available via `http://www.springer.co.uk/eWiC/DBPL5.html`).

Jacobs, C. E., A. Finkelstein, and D. H. Salesin (1995, August). Fast multiresolution image querying. In *Proceedings of SIGGRAPH 95*, pp. 277–286.

Jones, M. P. (1994a, May). The implementation of Gofer functional programming languages. Technical Report RR-1030, Yale University.

Jones, M. P. (1994b). *Qualified Types: Theory and Practice*. Cambridge University Press, Distinguished Dissertations in Computer Science.

Keene, S. E. (1989). *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS*. Addison Wesley.

Kirby, G. N. C. (1992). *Reflection and Hyper-Programming in Persistent Programming Systems*. Ph. D. thesis, University of St. Andrews.

Launchbury, J. (1995). Graph algorithm with a functional flavor. In J. Jeuring and E. Meijer (Eds.), *Adavanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques, Tutorial Text*, pp. 308–331. Lecture Notes in Computer Science 925, Springer-Verlag.

Launchbury, J. and S. L. Peyton Jones (1994, June). Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming language Design and Implementation*, pp. 24–35.

Libkin, L., R. Machlin, and L. Wong (1996, May). A query language for multidimensional arrays: Design, implementation, and query technique. In H. V. Jagadish and I. S. Mumick (Eds.), *Proceedings of the 1996 ACM SIGMOD Conference on Management of Data*, pp. 228–239.

Maier, D. and J. Stein (1987). Development and implementation of an object-oriented dbms. In B. Shriver and P. Wegner (Eds.), *Research Directions in Object-Oriented Programming*. MIT Press.



Matthes, F. and J. W. Schmidt (1995, September). Persistent threads. In J. Bocca, M. Jarke, and C. Zaniolo (Eds.), *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pp. 403–414.

McCarthy, D. R. and U. Dayal (1989, June). The architecture of an active data base management system. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pp. 215–224.

McNally, D. J. (1993). *Models for Persistence in Lazy Functional Programming Systems*. Ph. D. thesis, University of St. Andrews.

McNally, D. J. and A. J. T. Davie (1991, May). Two models for persistence in lazy functional programming systems. *SIGPLAN NOTICES 25*(5), 43–52.

Milner, R., M. Tofte, and R. Harper (1990). *The definition of Standard ML*. The MIT Press.

Moggi, E. (1989, June). Computational lambda-calculus and monads. In *Proceedings of Symposium on Logic in Computer Science*, pp. 14–23.

Motakis, I. and C. Zaniolo (1997, June). Temporal aggregation in active database rules. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pp. 440–451.

Naqvi, S. and S. Tsur (1989). *A Logical Language for Data and Knowledge Base*. Computer Science Press.

Nikhil, R. S. (1988). Functional databases, functional languages. In M. P. Atkinson, P. Buneman, and F. Bancilhon (Eds.), *Data Types and Persistence*, pp. 51–68.

Nikhil, R. S. (1990). The semantics of update in a functional database programming language. In F. Bancilhon and P. Buneman (Eds.), *Advances in Database Programming Language*, pp. 403–421.

Object Management Group (1997, August). The common object request broker: Architecture and specification, ver 2.1. Technical Report formal/97-09-01, Object Management Group.

Ohori, A. (1990). Representing object identity in a pure functional language. In *Proceedings of the Third International Conference on Database Theory*, pp. 41–55. Springer-Verlag.

Paepcke, A. (1988, August). PCLOS: A flexible implementation of CLOS persistence. In S. Gjessing and K. Nygaard (Eds.), *ECOOP'88: Proceedings of the Second European Conference on Object-Oriented Programming*, pp. 374–389. Lecture Notes in Computer Science 322, Springer-Verlag.

Paton, N., R. Cooper, H. Williams, and P. Trinder (1996). *Database Programming Languages*. Prentice Hall.

Paulson, L. C. (1996). *ML for the Working Programmer* (2nd. ed.). Cambridge University Press.

Peterson, J. and K. Hammonad, eds. (1996, May). *Report on the Functional Programming Language Haskell, Version 1.3*. `http://haskell.org`.

Peterson, J. and K. Hammonad, eds. (1997, April). *Report on the Functional Programming Language Haskell, Version 1.4*. `http://haskell.org`.

Peterson, J. C. (1994, May). Dynamic typing in Haskell. Technical Report RR-1022, Yale University.

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall.

Peyton Jones, S. L. and D. Lester (1992). *Implementing Functional Languages: A Tutorial*. Prentice-Hall.



Peyton Jones, S. L. and P. Wadler (1993, Jan.). Imperative functional programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 71–84.

Poulovassilis, A. and P. Kind (1990, March). Extending the functional data model to computational completeness. In *Proceedings of the International Conference on Extending Database Technology*, pp. 75–91. Lecture Notes in Computer Science 416, Sprinter-Verlag.

Poulovassilis, A., S. Reddi, and C. Small (1996, October). A formal semantics for an active functional DBPL. *Journal of Intelligent Information Systems 7*(2), 151–172.

Poulovassilis, A. and C. Small (1996, April). Algebraic query optimization for database programming languages. *VLDB Journal 5*(2), 119–132.

Reddi, A., A. Poulovassilis, and C. Small (1995, September). Extending a functional DBPL with ECA-rules. In T. Sellis (Ed.), *Proceedings of the Second International Workshop on Rules in Databases (RIDS-2)*, pp. 101–115. Lecture Notes in Computer Science 985, Springer-Verlag.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.

Seshadri, P., M. Livny, and R. Ramakrishnan (1997, Septermber). The case for enhanced abstract data types. In *Proceedings of the Thirty-third International Conference on Very Large Data Bases*, pp. 66–75.

Shipman, D. W. (1981, March). The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems 6*(1), 140–173.

Singhal, V., S. Kakkad, and P. Wilson (1992, September). Texas: An efficient, portable persistent store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pp. 11–33.

Small, C. (1993, December). A functional approach to database updates. *Information Systems 18*(8), 581–595.

Small, C. and A. Poulovassilis (1991, August). An overview of PFL. In *Proceedings of the Third International Workshop on Database Programming Languages*, pp. 96–111. Morgan Kaufmann.

Stonebraker, M., A. Jhingran, J. Goh, and S. Potamianos (1990, May). On rules, procedures, caching and views in data base systems. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pp. 281–290.

Sutton, D. and C. Small (1995, July). Extending functional database languages to update completeness. In C. A. Goble and J. A. Keane (Eds.), *Proceedings of the Thirteenth British National Conference on Databases*, pp. 12–14. Lecture Notes in Computer Science 940, Springer-Verlag.

Teorey, T. J. (1994). *Database Modeling & Design: The Fundamental Principles* (2nd ed.). Morgan Kaufmann.

Trinder, P. (1991). Comprehensions, a query notation for DBPLs. In P. Kanellakis and J. W. Schmidt (Eds.), *Proceedings of the Third International Workshop on Database Programming Languages*, pp. 55–68. Morgan Kaufmann.

Trinder, P. W. (1995). Data dependent concurrency control. In *Functional Programming, Glasgow 1994*, pp. 231–244. Springer-Verlag.

Wadler, P. (1987). List comprehensions. In *Peyton Jones (1987), Chapter 3*, pp. 127–138.

Wadler, P. (1989, January). How to make ad-hoc polymorphism less ad-hoc. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 60–76.

Wadler, P. (1990). Linear types can change the world! In M. Broy and C. B. Jones (Eds.), *Programming Concepts and Methods*. North Holland.



Wadler, P. (1992a, June). Comprehending monads. *Mathematical Structures in Computing Science 2*(4).

Wadler, P. (1992b, January). The essence of functional programming. In *Proc. ACM Symposium on POPL*, pp. 1–14.

Wadler, P. (1992c). Monads for functional programming. In M. Broy (Ed.), *Program Design Calculi, Proc. Marktoberdorf Summer School, Jul 30 – Aug 8, 1992*.

Widom, J. (1996). The Starburst active database rule system. *IEEE Transactions on Knowledge and Data Engineering 8*(4), 593–595.

Wilson, P. R. (1992, September). Uniprocessor garbage collection techniques. In *Proceedings of the 1992 International Workshop on Memory Management*, pp. 1–42.

Yale Haskell Group (1994, September). The yale haskell users manual. Technical report, Yale University.

# Appendix A

# Notes on the Implementation Details

The prototype has been built using Hugs 1.3 with Texas Persistent Store 0.5 under Solaris 2.5.1[1]. This chapter briefly explains the internal organization of these softwares, and then addresses the required modification. Note that we often refer to C-preprocessor macros in the following explanation. These values are defined and used internally by Hugs.

## A.1 Internal storage structure of Hugs

This section overviews some internal structures of Hugs to understand the modification points to port it on top of Texas Persistent Store. Readers interested in the details refer to (Jones 1994a) and the source code. Hugs comprises several internal stores: name dictionary, type dictionary, stack and heap. The name and type dictionaries give maps from symbols to definitions of top-level names and types respectively. Every definition is represented internally by a tree of cells. A cell is referred to by its address. For clarity, we denote these two concepts by using *cell-address* and *cell-value*, respectively, and use the term, *cell*, to denote the concept which comprises its address and value. The cell-address space is the set of signed integers. The negative address are used to locate cells in the heap area described below. Zero is used to represent special null pointer. The

[1]Solaris is a registered trademark of Sun Microsystems Inc.

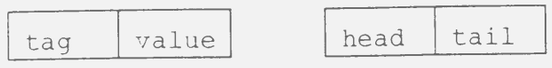



Figure A.1: Typical cell values.

positive cell-address are used to represent special boxed cells and tags. A typical boxed cell is a character value. For example, a character `'a'` is represented by a cell-address `CHARMIN+0x61` where `CHARMIN` is a C language macro value to give the minimum offset of the character value, and `0x61` is the ASCII code of `'a'`. Hence, the value of the cell is embedded in the address itself or *boxed*.

Another kind of boxed cells contain index values of tables. The language interpreter includes several tables to store type definitions and top-level names. An entry of these tables are referred to by an boxed index value. For example, suppose that a type definition as follows is stored in a name table entry $i$:

```
> type Natural = Int
```

Then, the value "`TYPMIN` + $i$" is the boxed cell of this entry, where `TYPMIN` is also a C-preprocessor macro.

As shown in these examples, the cell-value of a boxed cell is embedded in the address. However, there are many addresses that does not have cell-values. These values are called *tags* and are used to indicate sorts of cell-values.

A cell-value of *unboxed* cells are basically stored in the (fixed-length) heap area. The structure of a cell-value is shown in Fig. A.1. Each cell is represented by a pair of values[2]. Fig. A.1 (a) exhibits a tagged value where the left storage records the sort of the cell, and the right storage is used to record the value of that type. These definitions are used to construct special values such as type definitions and parsed expressions. For example,

[2]In the internal implementation, there are two arrays to store respectively the first values of cells and the second values of the cells.



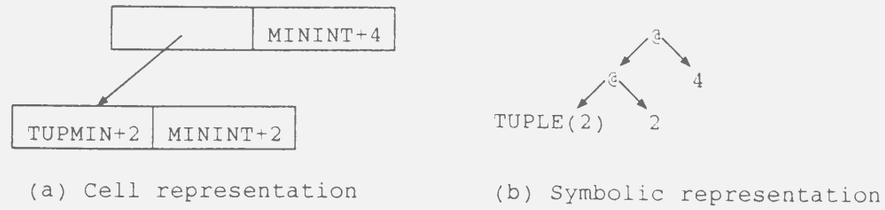



Figure A.2: Tree to represent `(2, 4)`

a cell-value having `POLY` as its first value is a value which represents a polymorphic type $\forall\alpha.\sigma$. The second value of the cell-value is a cell which in turn represents $\sigma$[3].

The cell-value structure shown in Fig. A.1 (b) is more prevailing in the system. For example, a tuple of integer value 2 and 4 are represented by a cell tree shown in Fig. A.2 (a), and corresponds to the symbolic tree shown in Fig. A.2 (b). In these figures, `TUPMIN` is the C-preprocessor macro to construct a tuple constructor, `TUPLE(2)` is the symbolic representation of this constructor, and the `@` mark is used to represent an pair or *application* of values.

There are two other storage types: the instruction storage and the flat space. Hugs executes expression after compilation, where the compile target is an abstract machine whose byte-code sequences are interpreted at run-time. Hence, after loading a script, the top-level values are compiled into byte-code sequences. These sequences are stored in the instruction storage, and the entry point is recorded in the top-level name table. The flat space is used to allocate *variable-length* records.

Lastly, we refer to garbage collection phases. The main heap area is maintained by a mark-sweep garbage collector. In the marking phase, the collector traverses all the cells reachable from prescribed roots such as the name table, and marks the live cells. In the second phase, in turn, the dead cells are collected to construct a list of free cells. On the other hand, the flat-space is maintained by a copying garbage collector. The flat

[3]In Hugs, polymorphic type also contains information called "kinds". However, we ignore it since it is not important in this context.

space is split into two spaces, *from-space* and *to-space*. Flat space cells are allocated from the from-space. In the garbage collection phase, live flat space values are moved to the to-space from the beginning, and after moving all the live values, the from-space and to-space are swapped. There is a complicated connection between the normal heap and the flat space, but we only show the the outline of the garbage collection phases:

- *Marking phase*: scans from the root cells to discriminate live normal help cells. This phase also analyzes the internal structure of each flat-space record so that the marking phase is invoked again from the cell values in the record.
- *Copying phase*: copies live flat-space cells to the to-space, and swaps them after that.
- *Sweep phase*: sweep the main heap cells to construct a free cell list.

## A.2 Using persistent store

As has been explained, there are several stores used to represent run-time environment: the tables to store various information, and the heaps. What we have to do to run Hugs on top of Texas Persistent Store is to modify the language processor so that these tables and heaps are allocated in a persistent storage, instead of the volatile memory area. The rest of this section describes such aspects that had to be required to be taken into account.

### A.2.1 Implementing module persistence

In contrast to a volatile programming environment, we must support *module persistence* (McNally and Davie 1991), which is to say that even after a module is discarded or reloaded, the reachable subexpressions must be remain on memory. This required the modification of the script management strategy of Hugs. In other words, allocating the heaps and information tables on a persistent area is not enough for supporting the module



persistency. This can be easily seen by considering the following program:

```
> f = transaction (
>       do p <- newDB Basic{ pName="B001", pCost=100, pMass=20,
>                            pUsedBy=[], pSuppliedBy=[] }
>          PartExt{parts=ps} <- getRootDB
>          setRootDB (PartExt{parts=p:ps}) )
```

Even after the evaluation of `f`, the newly generated labeled-record value may be a closure which points to a byte-code sequence via an internally generated name. The module persistence requires that even after the script defining this function is discarded, the internal closure remains valid. To make the software support the module persistency, the name information and byte-code sub-sequences have had to been moved to the flat space. After this modification, every name has two internal representations: an entry in the global name table to access it by its name, and another entry allocated in the flat space to record the details such as the type, classes, and instances. The global name table is used only in resolving names during compilation. After that, the name information is referred to via the pointer to the flat space.

### A.2.2 Flat-space garbage collection strategy

We also note the required modification in the flat-space management strategy for the purposes described in the previous subsection. Originally, the space is designed only to store dictionary values (Wadler 1989; Jones 1994b) that are *hidden* parameter values to implement overloaded functions. This poses a critical problem in the implementation of the name information and byte-code sequences in the flat area.

Fig. A.3 exhibits the run-time structure of a record in the flat space. The top-left cell in Fig. A.4, on the other hand, shows its status after the marking phase is completed. Notice that the pointer from the cell in the main heap to the flat space has disappeared. The liveness of the area in the flat space can be determined by seeing if its corresponding

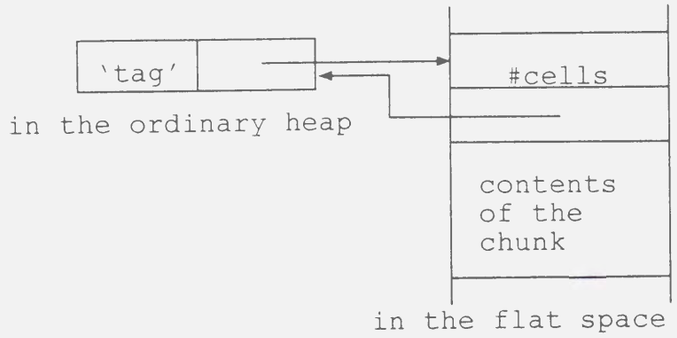



Figure A.3: Connection between the normal heap and the flat heap

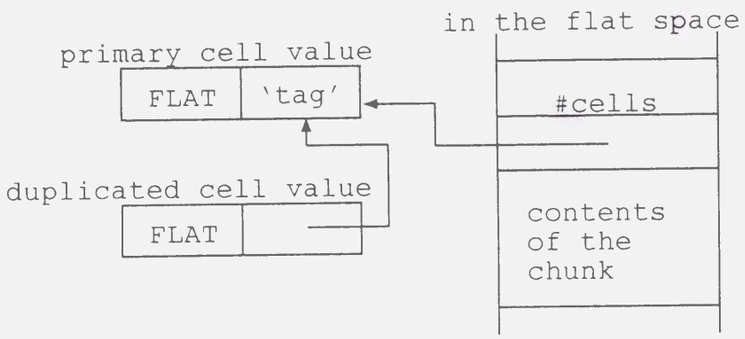



Figure A.4: Internal structure of a flat-space object (after "hacking")



cell-value is tagged with `FLAT`. Later in the copying phase, all the live records in the flat space are moved to the to-space. The disappeared pointer from the main heap to the flat space is recovered after the copying phase is completed.

Hugs is designed assuming *linearity* in the dictionary value usage. That is, no dictionary value is duplicated. This is not the case after the modification described in the previous section. To handle the duplication, the scan phase has been modified so that the duplicated cell value depicted at the bottom in Fig. A.4 records a pointer to the original cell value, instead of the original tag value. The duplicated cells are resumed in the sweep phase.

### A.2.3 Script management

Lastly, the script management had to be slightly modified. As noted above, in the original implementation, the script are loaded completely when required. The request is made by a user through the `:load`, `:append`, or `:project` command. The language processor automatically checks the modification of files to which any one of the requested files depends. Whenever such a file has been modified since the last loading time, it is automatically loaded again.

In the persistent environment, however, reloading file may modify a database schema or corrupt the database contents. To avoid such unexpected script reloading, the important scripts can be designated by the user by the `:lock` meta-command. Locked scripts are not reloaded even though it has been (accidentally) modified since the last loading time.

## A.3 Transaction management

The atomicity of transactions have been implemented through hidden global registers, and a trick using a *constant application form*.

### A.3.1 Background: constant applicative forms

Lazy evaluation is a technique to implement the call-by-name evaluation strategy. The key point in the evaluation is that a heap cell representing the same expression are shared, and whenever the expression is evaluated, the cell is updated with the evaluation result. This strategy is implemented in a straightforward way in normal subexpressions. The top level function, however, should be treated in a different way. As an example, consider the following simple top-level function

```
> v = 1 + 2
```

This value `v` is referred to by its name. That is, when `v` is used in a expression, its heap representation refers to the expression by the name cell for `v`. When the value of `v` is required to proceed evaluation, `v`'s definition, `1 + 2`, is reduced to 3 by the byte-code sequence associated with `v`, and the result 3 overwrites the the name cell. Hence, if another expression also refers to `v`, evaluation of v is performed again to get the expression result. Although the above example is simple, the top-level expression may be arbitrarily complex; indeed, it sometimes includes a mathematical sequence of values like

```
> fib = [1, 1] ++ [ fib ! (i-1) + fib ! (i-2) | i <- [2..]]
```

This values contains the Fibonacci sequence, and it would be useless to repeat computation of the list elements.

These values are called *constant applicative forms* (CAFs) because of its nature. The evaluation result of a CAF must affect all the cells referring to the CAF as noted just above. This aspect may be archived by compilation, linking (if any), or interpreter. The current Hugs implementation uses the interpreter approach. Whenever a CAF is evaluated at the first time, the result is cached in the name record. If the name is used again, the cached value is simply used by the interpreter to avoid the repeated evaluation of the value.



### A.3.2 Controlling database state initialization

The CAF mechanism is made use of to see if the database state initialization is required or not. Whenever a script is loaded in the language processor, a flesh name record is allocated for each top-level function. Therefore, the following code is enough to discriminate the need of initialization:

```
> shouldInitializeDB :: Ref (Bool)
> shouldInitializeDB = unsafePerformIO (newRef True)
```

where `unsafePerformIO` is a built-in function[4] which "wraps" an IO monad expression to the corresponding non-monad expression, and `newRef` generates a mutable variable for the I/O monad. This is a dangerous function which corrupts the single-threadedness of the I/O state transformers. We call this method "trick" because of this dangerous feature.

The compiler generates a CAF for this name. When the CAF is evaluated for the first time, the right-hand side of the equation is evaluated to generate an `IO` variable having `True`. This indicates initialization of the database state is required. After the database state is initialized, the following expression to flip the flag is executed in the bookkeeping code:

```
>          writeRef shoudlInitializeDB False
```

This part depends too much on the Hugs strategy for CAF management. This trick, however, is suitable for experimenting various database state management strategies implemented in different script files.

### A.3.3 Catch-and-throw

We must handle two atomicity for

- a transaction wrapped by the `transaction` function; and

[4]The original Hugs implementation does not include this function.

- a user-interaction session.

The first atomicity is treated by the versioning mechanism described Chapter 8. This subsection describes the second aspect. When a user invokes the persistent Haskell environment, (s)he can type in an arbitrary I/O expression which may include database transaction generated by `transaction`. During the evaluation of the I/O expression, the database state may be modified. However, we must catch the interruption of the expression evaluation which is invoked by typing `Ctr-C` (or something like that) during evaluation. Moreover, the I/O function and evaluation may incur evaluation errors such as opening non-existent files, and pattern-matching failure.

Handling such escape from evaluation phase to the topmost interaction session must be accompanied with database state restoring. Hugs internally uses *longjump* which is a catch-and-throw mechanism supported in C to directly return from a function nestedly called from another function to the designated point. The database state management routine is implemented in this catcher and thrower using persistent global variables. The first variable, named `gr0`, stores the current database state. The value is copied to another variable, `gr1`, before evaluating an expression typed-in by a user. Whenever the thrower executes the *longjump* to return to the catching point, the saved value in `gr1` is restored in the register `gr0`. If the expression typed in by the user is successfully executed, the contents of the `gr1` is simply discarded (to avoid space leak).

The catch-and-throw mechanism is implemented via the cooperation of the internal C routines to manipulate `gr0` and `gr1`, and the database state manipulation functions written in Haskell to abstract access to `gr0`.

# Appendix B

# Scripts for the Train Database

## B.1 Initialization for the Train Database

```
> module TrainInit where
> import Train
>
> type STime = String
> data TimeType = Arvl STime | Dprt STime | Stop STime | ArDp STime STime
> initdb = transaction initStationsAndTrains
>
> initStationsAndTrains =
>   do -- main stations
>      aberdeen     <- mkMain "Aberdeen"
>      inverness    <- mkMain "Inverness"
>      edinburgh    <- mkMain "Edinburgh"
>      glasgow      <- mkMain "Glasgow"
>      london       <- mkMain "London"
>      southampton  <- mkMain "Southampton"
>      -- local stations
>      nairn        <- mkLocl "Nairn" inverness
>      aviemore     <- mkLocl "Aviemore" inverness
>      kingussie    <- mkLocl "Kingussie" inverness
>      pitlochry    <- mkLocl "Pitlochry" inverness
>      perth        <- mkLocl "Perth" edinburgh
>      stirling     <- mkLocl "Stirling" edinburgh
>      falkirk      <- mkLocl "Falkirk" edinburgh
>      aberdour     <- mkLocl "Aberdour" edinburgh
>      banchory     <- mkLocl "Banchory" aberdeen
>      prestonpans  <- mkLocl "Prestonpans" edinburgh
>      northBerwick <- mkLocl "North Berwick" edinburgh
>      york         <- mkLocl "York" edinburgh
>      winchester   <- mkLocl "Winchester" southampton
```

```
>
>     ------------------
>     -- add train
>     ------------------
>     -- (1) Trains from and around Aberdeen.
>     --  aberdeen to edinburgh
>     mkTrain 22403101 [ (aberdeen   ,Dprt "07:20"),
>                        (edinburgh  ,Arvl "09:50") ]
>     mkTrain 22407101 [ (aberdeen   ,Dprt "10:00"),
>                        (edinburgh  ,Stop "13:00"),
>                        (york       ,Stop "15:00"),
>                        (london     ,Arvl "17:00") ]
>     mkTrain 22403102 [ (aberdeen   ,Dprt "14:00"),
>                        (edinburgh  ,Arvl "16:30") ]
>
>     -- aberdeen to inverness
>     mkTrain 22446301 [ (aberdeen   ,Dprt "08:00"),
>                        (inverness  ,Arvl "10:45") ]
>     mkTrain 22446302 [ (aberdeen   ,Dprt "12:00"),
>                        (inverness  ,Arvl "14:45") ]
>     mkTrain 22446303 [ (aberdeen   ,Dprt "19:00"),
>                        (inverness  ,Arvl "21:45") ]
>
>     -- aberdeen to banchory
>     mkTrain 22490101 [ (aberdeen   ,Dprt "08:30"),
>                        (banchory   ,Arvl "08:50") ]
>     mkTrain 22490102 [ (aberdeen   ,Dprt "12:00"),
>                        (banchory   ,Arvl "12:20") ]
>     mkTrain 22490103 [ (aberdeen   ,Dprt "18:00"),
>                        (banchory   ,Arvl "18:20") ]
>
>     -- (2) Trains from and around Inverness
>     mkTrain 46307101 [ (inverness  ,Dprt "06:30"),
>                        (edinburgh  ,Stop "10:30"),
>                        (york       ,Stop "12:30"),
>                        (london     ,Arvl "14:30") ]
>     mkTrain 46303101 [ (inverness  ,Dprt "08:40"),
>                        (aviemore   ,Stop "09:45"),
>                        (kingussie  ,Stop "11:00"),
>                        (perth      ,Stop "11:30"),
>                        (stirling   ,Stop "12:30"),
>                        (falkirk    ,Stop "12:40"),
>                        (edinburgh  ,Arvl "12:50") ]
>     mkTrain 46303102 [ (inverness  ,Dprt "10:15"),
```



```
>                        (edinburgh  ,Arvl "14:15") ]
>     mkTrain 46307102 [ (inverness  ,Dprt "14:00"),
>                        (edinburgh  ,Stop "18:00"),
>                        (london     ,Arvl "21:30") ]
>
>     mkTrain 46322401 [ (inverness  ,Dprt "09:00"),
>                        (aberdeen   ,Arvl "11:45") ]
>     mkTrain 46322402 [ (inverness  ,Dprt "12:00"),
>                        (aberdeen   ,Arvl "14:45") ]
>     mkTrain 46322403 [ (inverness,  Dprt "19:10"),
>                        (aberdeen,   Arvl "21:50") ]
>
>     mkTrain 46390101 [ (inverness  ,Dprt "08:30"),
>                        (nairn      ,Arvl "08:50") ]
>     mkTrain 46390102 [ (inverness  ,Dprt "18:00"),
>                        (nairn      ,Arvl "18:20") ]
>     mkTrain 46390103 [ (inverness  ,Dprt "19:10"),
>                        (nairn      ,Arvl "19:30") ]
>
>     mkTrain 46390201 [ (nairn      ,Dprt "06:00"),
>                        (inverness  ,Arvl "06:20") ]
>     mkTrain 46390202 [ (nairn      ,Dprt "08:00"),
>                        (inverness  ,Arvl "08:30") ]
>     mkTrain 46390203 [ (nairn      ,Dprt "18:00"),
>                        (inverness  ,Arvl "18:20") ]
>
>     -- Trains from and around Edinburgh
>     mkTrain 03146301 [ (edinburgh    ,Dprt "06:00"),
>                        (inverness    ,Arvl "10:00") ]
>     mkTrain 07146301 [ (london       ,Dprt "05:30"),
>                        (york         ,Stop "07:30"),
>                        (edinburgh    ,Stop "09:30"),
>                        (inverness    ,Arvl "13:30") ]
>     mkTrain 07146302 [ (london       ,Dprt "10:00"),
>                        (york         ,Stop "12:10"),
>                        (edinburgh    ,ArDp "14:20" "14:40"),
>                        (falkirk      ,Stop "15:00"),
>                        (stirling     ,Stop "15:30"),
>                        (perth        ,Stop "16:30"),
>                        (pitlochry    ,Stop "17:30"),
>                        (kingussie    ,Stop "18:00"),
>                        (aviemore     ,Stop "18:15"),
>                        (inverness    ,Arvl "19:00") ]
>     mkTrain 03146303 [ (edinburgh    ,Dprt "17:20"),
```

```
>                       (inverness    ,Arvl "20:00") ]
>
>     mkTrain 03122401 [ (edinburgh   ,Dprt "08:00"),
>                       (aberdeen    ,Arvl "10:45") ]
>     mkTrain 07122401 [ (london      ,Dprt "08:00"),
>                       (york        ,Stop "10:00"),
>                       (edinburgh   ,Stop "12:00"),
>                       (aberdeen    ,Arvl "14:45") ]
>     mkTrain 07122402 [ (edinburgh   ,Dprt "15:00"),
>                       (aberdeen    ,Arvl "17:45") ]
>     mkTrain 03122403 [ (edinburgh   ,Dprt "19:00"),
>                       (aberdeen    ,Arvl "21:45") ]
>
>     mkTrain 03104101 [ (edinburgh   ,Dprt "07:00"),
>                       (glasgow     ,Arvl "08:00") ]
>     mkTrain 03104102 [ (edinburgh   ,Dprt "10:40"),
>                       (glasgow     ,Arvl "11:40") ]
>     mkTrain 03104103 [ (edinburgh   ,Dprt "14:40"),
>                       (glasgow     ,Arvl "15:40") ]
>     mkTrain 03104104 [ (edinburgh   ,Dprt "19:00"),
>                       (glasgow     ,Arvl "20:00") ]
>
>     mkTrain 03107101 [ (edinburgh   ,Dprt "06:00"),
>                       (york        ,Stop "08:00"),
>                       (london      ,Arvl "10:00") ]
>
>     mkTrain 03190101 [ (edinburgh   ,Dprt "08:30"),
>                       (aberdour    ,Arvl "08:50") ]
>     mkTrain 03190102 [ (edinburgh   ,Dprt "18:20"),
>                       (aberdour    ,Arvl "18:40") ]
>
>     mkTrain 03190201 [ (aberdour    ,Dprt "07:00"),
>                       (edinburgh   ,Arvl "07:30") ]
>     mkTrain 03190202 [ (aberdour    ,Dprt "14:00"),
>                       (edinburgh   ,Arvl "14:20") ]
>
>     mkTrain 03190301 [ (edinburgh   ,Dprt "08:10"),
>                       (prestonpans ,Stop "08:35"),
>                       (northBerwick,Arvl "09:30") ]
>     mkTrain 03190302 [ (edinburgh   ,Dprt "13:40"),
>                       (prestonpans ,Stop "14:05"),
>                       (northBerwick,Arvl "15:00") ]
>     mkTrain 03190303 [ (edinburgh   ,Dprt "15:00"),
>                       (prestonpans ,Stop "15:25"),
```



```
>                       (northBerwick,Arvl "16:20") ]
>     mkTrain 03190304 [ (edinburgh   ,Dprt "18:20"),
>                       (prestonpans ,Stop "18:25"),
>                       (northBerwick,Arvl "19:40") ]
>
>     mkTrain 03190401 [ (northBerwick,Dprt "06:10"),
>                       (prestonpans ,Stop "07:25"),
>                       (edinburgh   ,Arvl "07:45") ]
>     mkTrain 03190402 [ (northBerwick,Dprt "08:45"),
>                       (prestonpans ,Stop "09:15"),
>                       (edinburgh   ,Arvl "10:15") ]
>     mkTrain 03190403 [ (northBerwick,Dprt "11:20"),
>                       (prestonpans ,Stop "11:55"),
>                       (edinburgh   ,Arvl "12:45") ]
>     mkTrain 03190404 [ (northBerwick,Dprt "12:50"),
>                       (prestonpans ,Stop "13:15"),
>                       (edinburgh   ,Arvl "14:25") ]
>
>    -- Trains from and around Glasgow
>     mkTrain 04103101 [ (glasgow     ,Dprt "09:00"),
>                       (edinburgh   ,Arvl "09:50") ]
>     mkTrain 04103102 [ (glasgow     ,Dprt "10:30"),
>                       (edinburgh   ,Arvl "11:30") ]
>     mkTrain 04103103 [ (glasgow     ,Dprt "13:10"),
>                       (edinburgh   ,Arvl "14:10") ]
>     mkTrain 04103104 [ (glasgow     ,Dprt "17:15"),
>                       (edinburgh   ,Arvl "18:15") ]
>
>     mkTrain 04107101 [ (glasgow     ,Dprt "07:00"),
>                       (london      ,Arvl "11:00") ]
>     mkTrain 04107102 [ (glasgow     ,Dprt "10:10"),
>                       (london      ,Arvl "14:30") ]
>     mkTrain 04107103 [ (glasgow     ,Dprt "12:30"),
>                       (london      ,Arvl "17:10") ]
>     mkTrain 04107104 [ (glasgow     ,Dprt "18:00"),
>                       (london      ,Arvl "22:00") ]
>
>     mkTrain 07104101 [ (london      ,Dprt "07:00"),
>                       (glasgow     ,Arvl "11:00") ]
>     mkTrain 07104102 [ (london      ,Dprt "10:00"),
>                       (glasgow     ,Arvl "14:00") ]
>     mkTrain 07104103 [ (london      ,Dprt "18:30"),
>                       (glasgow     ,Arvl "22:30") ]
>
```

```
>      mkTrain 07103101 [ (london       ,Dprt "18:00"),
>                         (edinburgh    ,Arvl "22:00") ]
>
>      mkTrain 07170301 [ (london       ,Dprt "07:05"),
>                         (southampton  ,Arvl "08:20") ]
>      mkTrain 07170302 [ (london       ,Dprt "14:50"),
>                         (southampton  ,Arvl "16:15") ]
>      mkTrain 07170303 [ (london       ,Dprt "17:20"),
>                         (southampton  ,Arvl "18:45") ]
>      mkTrain 07170304 [ (london       ,Dprt "20:20"),
>                         (southampton  ,Arvl "21:35") ]
>
>      mkTrain 70307101 [ (southampton  ,Dprt "08:00"),
>                         (london       ,Arvl "09:00") ]
>      mkTrain 70307102 [ (southampton  ,Dprt "08:40"),
>                         (london       ,Arvl "09:50") ]
>      mkTrain 70307103 [ (southampton  ,Dprt "12:00"),
>                         (london       ,Arvl "13:00") ]
>      mkTrain 70307104 [ (southampton  ,Dprt "19:00"),
>                         (london       ,Arvl "21:15") ]
>
>      mkTrain 70390101 [ (southampton  ,Dprt "08:30"),
>                         (winchester   ,Arvl "08:50") ]
>      mkTrain 70390102 [ (southampton  ,Dprt "15:15"),
>                         (winchester   ,Arvl "15:40") ]
>      mkTrain 70390103 [ (southampton  ,Dprt "18:00"),
>                         (winchester   ,Arvl "18:20") ]
>      mkTrain 70390104 [ (southampton  ,Dprt "19:10"),
>                         (winchester   ,Arvl "19:30") ]
>      mkTrain 70390105 [ (southampton  ,Dprt "21:00"),
>                         (winchester   ,Arvl "21:30") ]
>
>      mkTrain 07390201 [ (winchester   ,Dprt "06:45"),
>                         (southampton  ,Arvl "07:30") ]
>      mkTrain 07390202 [ (winchester   ,Dprt "07:45"),
>                         (southampton  ,Arvl "08:25") ]
>      mkTrain 07390203 [ (winchester   ,Dprt "14:00"),
>                         (southampton  ,Arvl "15:00") ]
>
>      return ()
>
>
>    where
>      mkMain name
```



```
>        = newDB MainStation{stName=name, stStops=[]}
>    mkLocl name mn
>      = newDB LoclStation{stName=name, stStops=[], stNearestMn=mn}
>    mkTrain no visits
>      | ( case snd (head visits) of
>            Dprt _ -> True
>            _      -> False ) &&
>        ( case snd (last visits) of
>            Arvl _ -> True
>            _      -> False ) &&
>        ( if length visits >= 2 then
>            foldr (&&) True
>              [ case sched of {
>                    Stop _ -> True; ArDp _ _ -> True; _ -> False }
>                | (_, sched) <- take (length visits - 2) (tail visits) ]
>          else True )
>      = newDB Train{trNo=no, trVisits=visits'}
>      where visits'
>              = [ (st, mkArr sched, mkDep sched ) | (st,sched) <- visits ]
>            mkArr (Dprt time)    = Nothing
>            mkArr (Stop time)    = Just (mkTime time)
>            mkArr (ArDp time _) = Just (mkTime time)
>            mkArr (Arvl time)    = Just (mkTime time)
>            mkDep (Dprt time)    = Just (mkTime time)
>            mkDep (Stop time)    = Just (mkTime time)
>            mkDep (ArDp _ time) = Just (mkTime time)
>            mkDep (Arvl time)    = Nothing
```

## B.2 Searching Routes in the Train Database

```
> module TrainQuery2 where
> import Train
>
> findroute from to
>   = do db <- tran getDB
>        --- check routes in this database
>        let StationMap stmap = getRoot(db)
>            ss1 = [ sid | (name, sid) <- stmap, name == from ]
>            ss2 = [ sid | (name, sid) <- stmap, name == to   ]
>
>        --- check if the stations exist.
>        do when (ss1 == []) (
>             error ("Station \"" ++ from ++ "\" does not exist."))
>           when (ss2 == []) (
>             error ("Station \"" ++ to ++ "\" does not exist."))
>
>        --- dereference station values
>        let s1 = head ss1
>            s2 = head ss2
>            station1 = readRef(db) s1
>            station2 = readRef(db) s2
>
>        --- check if any train stops at the station.
>        do when (stStops station1 == []) (
>             error ("There is no train from \"" ++ from ++ ".\""))
>           when (stStops station2 == []) (
>             error ("There is no train from \"" ++ from ++ ".\""))
>
>        --- ... doing the job....
>        putStr ("... searching routes from " ++ from ++ " to " ++ to ++ "\n")
>
>        ---- find the routes.
>        let rts
>             = dofindroute db
>                 [[(s1, tid, arr, dep)] | (tid, arr, dep) <- stStops station1
>                 s2
>
>        --- print the computed routes.
>        printRoutes db rts
>        return ()
>
> printRoutes db rts
>   = sequence [
```



```
>       do putStr (take 40 (repeat '-') ++ "\n")
>          sequence [
>              do let stname = show14s (stName (readRef(db) sid))
>                     trname = show8d  (trNo   (readRef(db) tid))
>                     showTime time = case time of
>                                       Nothing -> "--:--"
>                                       Just tm -> show tm
>                     arrtm  = showTime arr
>                     deptm  = showTime dep
>                 putStr (stname ++ ": "
>                           ++ arrtm ++ ", " ++ deptm ++ ": " ++ trname ++ "\n")
>           | (sid, tid, arr, dep) <- reverse route ]
>       | route <- rts ]
>  where show14s s | length s < 14 = s ++ (take (14 - length s) (repeat ' '))
>                  | otherwise     = s
>        show8d x | x > 10000000 = show x
>                 | otherwise    = "0" ++ show x
>
> dofindroute ::
>   Database -> [[(DBRef Station, DBRef Train, Maybe Time, Maybe Time)]]
>            -> DBRef Station
>            -> [[(DBRef Station, DBRef Train, Maybe Time, Maybe Time)]]
>
> dofindroute db [] to
>   = []
> dofindroute db routes to
>   = reached ++ findMore
>   where reached    = [ path | path@((sid, _, _, _):_) <- routes, sid == to ]
>         notReached = [ path | path@((sid, _, _, _):_) <- routes, sid /= to ]
>         nextRoutes = do path <- notReached
>                         findNext db path to
>         findMore = dofindroute db nextRoutes to
>
> --- search the next stop: cyclic route is not allowed;
> findNext ::
>    Database -> [(DBRef Station, DBRef Train, Maybe Time, Maybe Time)]
>             -> DBRef Station
>             -> [[(DBRef Station, DBRef Train, Maybe Time, Maybe Time)]]
> findNext db [] to
>   = []
> findNext db path@((sid, tid, arrive, depart):_) to
>     -- has arrived at the destination
>   | sid == to
>   = [path]
```

```
>   --- departs from this station.  Add the next stop.
>   | arrive == Nothing || length path == 1
>   = [ (sid', tid, arr', dep'): path
>       | (sid', arr', dep') <- nextStop sid tid,
>         not (sid' `elem` map (\(x,_,_,_) -> x) path) ]
>
>   --- arrives here but requires further travel
>   | otherwise
>   = -- (1) with the same train.
>     [ (sid', tid, arr', dep'): path
>       | (sid', arr', dep') <- nextStop sid tid,
>         not (sid' `elem` map (\(x,_,_,_) -> x) path) ]
>    ++
>     -- (2) via another train.
>     [ (sid2, tid1, arr2, dep2):(sid, tid1, arr1, dep1): path
>       | (tid1, arr1, dep1@(Just dtime1)) <- stStops (readRef(db) sid),
>         tid1 /= tid,
>         dtime1 - atime <= mkTime "01:00",
>         dtime1 - atime >= mkTime "00:10",
>         (sid2, arr2, dep2) <- nextStop sid tid1,
>         not (sid2 `elem` map (\(x,_,_,_) -> x) path) ]
>  where
>    Just atime = arrive
>    Just dtime = depart
>    nextStop sid tid
>     = let Train{trNo,trVisits} = readRef(db) tid
>           sched = dropWhile (\(sid', _, _) -> sid' /= sid) trVisits
>       in if length sched == 0
>          then error ("dofindroute.nextStop: train " ++ show trNo ++
>            " does not stop at \"" ++ stName (readRef(db) sid) ++ "\".\n")
>          else take 1 (tail sched) --- maybe [], if no next stops.
```